Pre Silicon RTL Verification of Mixed Signal IPs

A Thesis Submitted in Partial Fulfilment of the Requirement for the Award of the Degree of

MASTER OF TECHNOLOGY

in VLSI Design

Submitted By
SHRUTI KHAJURIA
601662019

Under Supervision of
Dr. MOHIT AGARWAL
Assistant Professor
And
Mr. NITIN CHAUDHARY
Engineering Manager
Intel Technology India Pvt. Ltd. Bangalore

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT
THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB
JUNE, 2018

INTEL TECHNOLOGY INDIA PVT. LTD.
Bangalore – 560103, Karnataka, India

Date: June 11, 2018

CERTIFICATE

This is to certify that **Shruti Khajuria (Regn. No. 601662019)**, a student of M.Tech.(VLSI Design), **Thapar Institute of Engineering and Technology, Patiala** has successfully completed one year (August 2017 – June 2018) internship programme in **Intel Technology India Pvt. Ltd, Bangalore.** Her title of dissertation is "**Pre-Silicon RTL Verification of Mixed Signal IPs**".

During the period of her internship programme, she was punctual and hardworking.

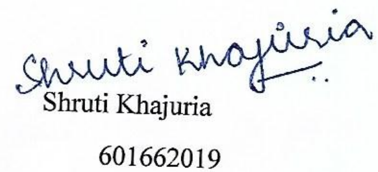
I wish her every success in life.

Nitin Chaudhary
Engineering Manager
MIG, India

DECLARATION

I, Shruti Khajuria, hereby declare that the work presented in this thesis entitled "Pre-Silicon RTL Verification of Mixed Signal IPs" in partial fulfillment of the requirement for the award of degree of Master of Technology (VLSI Design) submitted at ECED, Thapar Institute of Engineering and Technology (Deemed to be University), Patiala, is an authentic record of work carried out under supervision of Dr. Mohit Agarwal, Assistant Professor, ECED, Thapar Institute of Engineering and Technology, from June 2017 to June 2018. The matter presented in this has not been submitted either in part or full to any other university or institute, for the award of any other degree.

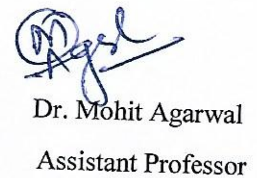
Date: 15 'June 18



Shruti Khajuria

601662019

It is certified that the above statement made by the student is best to my knowledge and belief.



Date: 15 June '18

Dr. Mohit Agarwal

Assistant Professor

ACKNOWLEDGEMENT

I would like to express my sincere thanks to Dr. Alpana Agarwal **,**Head of Department, ECED, Thapar Institute of Engineering and Technology (Deemed to be University), Patiala, for giving me the opportunity to do a yearlong project in Intel India Pvt. Ltd. I owe my profound gratitude to Dr. Mohit Agarwal, Assistant Professor, ECED, Thapar Institute of Engineering and Technology (Deemed to be University), Patiala, for his immense knowledge, continuous guidance and support. I express my sincere thanks to Mr. Nitin Chaudhary (Engineering Manager), Mr. B. P. Ganesh (Verification Engineer), Mr. Raja Vignesh (Technical Verification Lead), Intel Pvt. Ltd., who in spite of being extremely busy with their schedules, took out time to guide and help me with my project. I am also grateful to my colleagues for providing an amicable work environment.

Finally, I would also like to thank my family and friends for their patience and unwavering support during the course of this project.

Shruti Khajuria

(601662019)

ABSTRACT

Validation remains an unavoidable, yet truly difficult and prolonged part of IP configuration and manufacture process. With shortening timelines and expanding time-to-market necessity, IP producing houses are compelled to empty more and more resources into validation. Hence, validation remains an indispensable and essential period of today's IP design and integration procedure. Validation techniques can be partitioned into two stages: Pre-Silicon Validation and Post-Silicon Validation.

For modern ICs with ever increasing RTL complexity, it is becoming very difficult to test all the design specifications using conventional verification techniques having test-benches. There is always a potential for subtle interactions between the various components in a design and because of the design complexity, it is very difficult to confine a realistic set of possible interactions with the help of simulation. Due to this reason, there is an increasing demand for formal property verification. Formal Property Verification consists of exploring all states and transitions in the design by using efficient and domain specific abstraction techniques. These techniques keep into consideration the entire set of possible states in a single operation and hence reduce the human effort as well as the computation time. Pre Silicon verification is not enough to assure complete IP verification. Post silicon support is also required to debug the errors that might have escaped the pre-silicon verification process. Moreover, post silicon verification is done on the manufactured chip, operating at real world frequencies, such that the behavior of the design in the actual environment in which it is intended to be used can be checked.

TABLE OF CONTENTS

Sr. No.	Chapter Title	Page No.
	Certificate	*ii*
	Declaration	*iii*
	Acknowledgement	*iv*
	Abstract	*v*
	List of Figures	*vi*
	List of Tables	*vii*
	List of Abbreviations	*viii*
Chapter 1	**Introduction**	***2-5***
	1.1 Background	2
	1.2 Motivation for the project	3
	1.3 Objective of the project	4
	1.4 Organization of the report	4
Chapter 2	**Literature Review**	***5-15***
Chapter 3	**Verification Methodology**	***16-29***
	3.1 Open Verification Methodology	16
	3.2 System Verilog Assertions	20
	3.3 Formal verification and simulation	27
Chapter 4	**Register Space Access**	***30-38***
	4.1 Introduction	30
	4.2 Common Register Interface	30
	4.3 JTAG Protocol	31
	4.3.1 Introduction to JTAG	31
	4.3.2 TAP State Machine	32
	4.3.3 JTAG Registers	35
	4.3.4 JTAG Instructions	35
	4.3.5 Resetting The TAP	36

4.3.6 JTAG Registers 36

4.4 JTAG CRI Glue logic 38

Chapter 5 Post Silicon Support........ *39-43*

5.1. Introduction 39

5.2 Design for x (DFx) 40

5.3 Monitor port routing 41

5.3.1 Motivation for JTAG 41

5.3.2 Signal Routing 42

Chapter 6 Results and Observations........ *44-49*

6.1 Assertion based verification 44

6.2 Tests for post silicon support........ 47

6.3 Regression analysis 47

Chapter 7 Conclusion and Future Scope 50

REFERENCES........51

LIST OF TABLES

Table 3.1 Comparison between FV and conventional verification 26

LIST OF FIGURES

Figure 2.1 Transaction Based Verification 5
Figure 2.2 ASIC Verification Methodology 7
Figure 2.3 Randomized Test Vector Verification 7
Figure 2.4 Assertions provide temporal domain for functional coverage 8
Figure 2.5 Selecting effective regression test cases 13
Figure 3.1 Open Verification Methodology 15
Figure 3.2 System Verilog Assertions in Test Environment 17
Figure 3.3 Hierarchy of Assertion Layer 19
Figure 3.4 Property to be validated 20
Figure 3.5 Verilog code to verify the property 21
Figure 3.6 SV Assertion to Verify the Property 21
Figure 3.7 Improved Observability in Assertions 22
Figure 3.8 Assertions for Hardware Emulation 23
Figure 3.9 Assertions and Open Verification Library for different uses 23
Figure 3.10 RTL Formal PropertyVerification 25
Figure 4.1 Register Space Access 28
Figure 4.2 Interconnection of Various Test Devices 29
Figure 4.3 TAP Architecture 29
Figure 4.4 JTAG Finite State Machine 30
Figure 4.5 JTAG Register Operation 34
Figure 4.6 JTAG and CRI glue Logic 35
Figure 5.1 Verification Flow 36
Figure 5.2 Verification Techniques 37
Figure 5.3 Bed of Nails Technique 39
Figure 5.4 Test Flow for Signal Routing 40
Figure 6.1 Control Signals in JTAG 41
Figure 6.2 Waveforms of a Failing Assertion 42
Figure 6.3 Assertions Results in Formal Tool Checker 43
Figure 6.4 Significance of Cover Property 43
Figure 6.5 Simulation Waveforms of Routing Tests 44
Figure 6.6 Regression Analysis 45

LIST OF ABBREVIATIONS

AVM: Advanced Verification Methodology

APB: Advanced Peripheral Bus

AMBA: Advanced Microcontroller Bus Architecture

AMS: Analog Mixed Signal

ASIC: Application Specific Integrated Circuit

ATE: Automatic Test Equipment

ATPG: Automatic Test Pattern Generation

BERT: Bit Error Rate Tester

CMOS: Complementary Metal Oxide Semiconductor

CRI: Common Register Interface

DFD: Design For Debug

DUT: Design Under Test

DUV: Design Under Verification

FPV: Formal Power Verification

HDL: Hardware Description Language

IP: Intellectual Property

ISS: Instruction Set Simulator

ISTA: Internal Signal Trace Analysis

JTAG: Joint Test Action Group

OVM: Open Verification Methodology

RTL: Register Transfer Level

HSSI: High-Speed Serial Interface

TLM: Transaction Level Modeling

URM: Universal Reuse Methodology

UVM: Universal Verification Methodology

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

Verification is the process of validating whether the product that has been designed meets the required specifications or not. It is the process by which an engineer determines that the design intention is fully met with its functional expectations in its implementation. Verification consumes about 70% of the design effort, and the number of verification engineers can be up to twice the number of RTL design engineers. The process of verification can start from the time when the design architectures and micro architectures are defined. The main goal of verification is to confirm the functional correctness of the design before the tape out. After the tape out, screening of manufactured chips is done, which is called post silicon verification. It checks the chips for faults and defects which might have been resulted from the manufacturing errors. Additional hardware is also incorporated in the design so as to facilitate the debug process on the manufactured chip.

Verifying the functional property of current VLSI design is done using test benches. But as the increase in complexity of design with power intent no matter how smart our testbench is we may miss some functional coverage in the design (50 to 60% time is consumed in functionality testing). So instead of testing design under test (DUT) with test benches it's better to use formal methods. The formal property is a critical element in the development of today's complex digital designs. The reality is that no matter how long you simulate or how intelligent the testbench is, validating the design intent through simulation is inherently incomplete. Corner-case bugs are artifacts of simulation since they are not detected due to the non-exhaustive nature of simulation based verification.

Verifying the design and finding and fixing the issues at early stage of the design cycle makes the manufacturer more reliable. Ideally, all the bugs should be detected in early stage of pre silicon and there should be zero probability that issue gets into post silicon validation. However, post silicon validation is an integral part of the design cycle. As soon as the first silicon is out, tests are run on it to ensure proper functional correctness. In the post silicon environment, the test chip actually runs in the real conditions, on real frequencies. This is very different from pre silicon phase where simulations are run on RTL. Post silicon tests require additional logic added to the design to aid the post silicon debug process. This additional design structure helps in root causing the errors in post silicon, where simulation is not an option to trace back the failures.

There are two main goals of validation testing: first, to thoroughly test the functionality of the chip; second, to structure the code and plan the test environment to quickly and efficiently debug, isolate, and report problems and issues that come up in the lab while testing. This means putting together an

organized system of debug statements, logic analyzer triggers, event logging, and so on. There are three types of validation tests: directed diagnostic tests, stress tests, and real-world tests. Although there is some duplication in the functionality tested, these types represent different test dimensions that are all needed to ensure the quality of the silicon device.

1.2 MOTIVATION FOR THE PROJECT

Formal Property Verification (FPV) fully validates block-level properties and high level requirements. It enables exhaustive and complete verification and provides rapid bug detection as well as end-to-end full proofs of expected design behavior. FPV is a mathematical algorithm which will always try to find the failure case of design.

With the formal mathematical tools, the correctness of the design can be proved or disproved. A Finite State Machine can be visualized as a state transition graph in which the states are labeled with the values of various elements in the design. Properties about the system are expressed as formulas in temporal logic of which the state transition system is to be a “model". Model checking consists of traversing the graph of the transition system and of verifying that it satisfies the formula representing the property, i.e., the system is a model of the property.

With its powerful analysis capabilities and ease of use, it is ideal for early-stage bug hunting as well as for ensuring the highest confidence possible in design functionality via end-to-end full proofs. Low power checks are also possible with formal verification methods where properties are written in System Verilog.

1.3 OBJECTIVE OF THE PROJECT

The objective of the project is to verify the register access of the internal configuration registers in the PHY (Physical Layer in the OSI model) by JTAG protocol with the help of a formal verification tool. The verification methodology ensures that proper read and write of the internal configuration registers is taking place, in compliance with the underlying protocols. Also, post silicon support will be provided for additional visibility of deep RTL signals in the post silicon environment after the manufacture. This visibility helps in post silicon debug process, since otherwise the internal signals are not observable in pot silicon environment, and hence the debug process to root cause the underlying issue is very difficult. This support will be added in form of additional code inserted in the RTL to make the routing of these internal signals to the observation pads possible.

1.4 ORGANIZATION OF THE REPORT

This Chapter gives the overview of the project, introduction to formal property verification and the importance of post silicon debug support.

Chapter 2 comprises of the literature survey done in the domain of pre silicon as well as post silicon RTL verification. The objective behind verification is studied and of the growth of verification techniques and methodologies is analyzed.

Chapter 3 elucidates the verification methodologies used in this project. Open Verification Methodology and System Verilog Assertions are presented.

Chapter 4 defines how the internal register space is accessed and the corresponding glue logic between JTAG and CRI has been discussed.

Chapter 5 gives the techniques to aid post silicon debug. It includes the methodology adopted to add additional test hardware to root cause errors in manufactured silicon.

Chapter 6 presents the results and discussions on the System Verilog Assertions written to validate the register access protocols, post silicon tests to aid silicon debug and the regression testing technique used for continuous evaluation.

Chapter 7 concludes the study on pre silicon RTL verification and future scope of the work done is discussed.

CHAPTER 2

LITERATURE REVIEW

In the early 1990s, the system design communities underwent tremendous productivity gains in gate-level design as engineers embraced synthesis technology [1]. However, this led to the increase in the complexity in the verification of the design. To keep up with the increasing complexity, RTL coding style was introduced which optimized the verification flow. An RTL coding methodology that can be easily verified allows the design engineer to achieve higher quality of verification with increased coverage in minimal time. It leads to a clarified design intent and creates space for enhancing the support of multiple EDA tools for the emerging verification flows and processes. As a result, there is a significant reduction in the development time to market with increased verification coverage. Today, with the availability of reliable synthesizers and application of synchronous design techniques, the lowest level of detail that designers must consider is register transfer level, RTL [2]. Functionally verifying a design refers to comparing the designer's intent with the behaviour that is observed from the design so that their equivalence can be determined. The tern design is usually referred to as the DUV, the Design Under Verification. The designer's intent is captured in a reference model, which can be a set of assertions which describe a certain protocol, a golden model that contains a unique algorithm, or a document that describes the expected behaviour of the design. Proper verification of a design not only determines the equivalence between the design intent and the design under verification, but also confirms that the coverage with respect to the portion of the design that has been verified is also satisfactory. A system model of the design is developed using the specification of the functional intent by the designer. It is developed by incorporating the component elements of the design at system level, in a system level language, like C, C++, etc. [3].The system level model is verified by developing a system level test bench. The design is then decomposed into further abstraction levels, like architectural, HDL, netlist models.

Registers, interconnection between registers and the computation necessary to modify the values of these registers according to needs of the design, constitute the primary components of a design that is represented at the Register Transfer Level, or the RTL abstraction level. The values of these registers are subject to the change in the clock associated with them, and all the combinational logic that drives the values of these registers can be expressed in the form of Boolean as well as algebraic expressions. RTL bridges the hardware and the software world, as the components that are used to define the RTL are essentially identifiable as hardware components such as registers, wires, clocks, but at the same time, the way in which the RTL is written, using combinational expressions, is similar to that of typical programming languages such as C, Java, etc. The process by which an RTL design is built constitutes of programming the code which represents the behavior of the design. This is done

with the help of a specialized language for hardware description, or the HDL. The test benches that are developed to check the functional correctness of the design are also hardware aware, since their job is to analyze the hardware and respond to any changes that take place in it. It is very much appropriate to apply software building techniques to test benches because in their form, they are essentially software. Software construction is at the center of the modern verification methodologies, like OVM, UVM. Software construction is done on the principles of object oriented programming, code libraries and refactoring, testing strategies, and so on.

Transaction Level Modeling (TLM), a methodology based upon such abstraction, has proven revolutionary values in bringing software and hardware teams together using the unique reference model; resulting in dramatic reduction of time-to-market and improvement of SoC design quality[4]. The transaction based verification methodology raises the verification effort to a higher abstraction level and thus makes functional verification of RTL descriptions using simulation more effective [5].

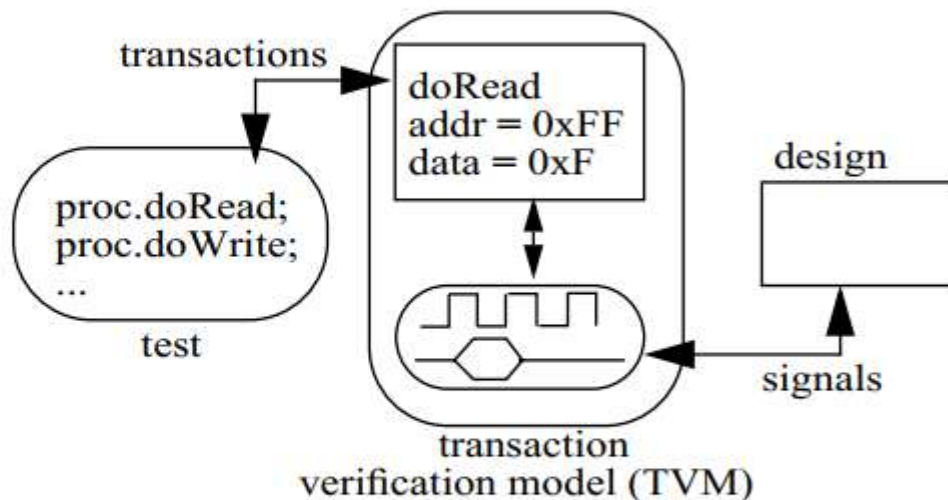


Figure 1.1 Transaction Based Verification [4]

This results in reusability of the components designed in a testbench. The information is represented in terms of transactions between the various components in the testbench which improves the coverage analysis and well as debugging, rather than signals and waveforms, as in conventional methodologies. The methodology depends on separating the testbench into different layers, the topmost constituting of the tests which direct the activity on the design under test to validate its functionality. The bottommost layer is the Transaction Verification Model (TVM) which is responsible for converting the transaction level requests made by the tests into the pin level activity

based on the signal level protocols on the design interface. With the separation of the layers, new tests can easily be added to the topmost layer, and the defining protocols in the TVM remain unchanged.

Pre Silicon Verification refers to the validation done on the design before it goes for actual silicon manufacturing. It is critical to ensure product quality and detecting the bugs in the early stages of the development cycle gives the product a shorter time to market. Detecting bugs and fixing the issues in Post silicon environment is a costly affair, and leads to a compromise in the product quality. Current pre-silicon validation efforts are usually distributed among various teams in the semiconductor industry. These teams are primarily design teams, tool maintainers, tool developers and validation teams. There needs to be proper communication between these teams so as to ensure that no bugs escape the pre silicon validation efforts.

Due to the complexity of the SoC design, it is imperative for a designer to check both hardware and software functionality as well as performance at the early stage of development. Native System C assertion mechanism is used to generate temporal assertions for higher abstraction models [6]. These assertions check the design behavior on two different levels, one at the event sample which governs the timing requirements, and the other which consists of event based sequence. These work on the occurrence of events, without the aid of any clocks.

Verification Methodologies help minimize the time that goes into the entire verification process of a design under test. These methodologies define standards which allow interoperability between verification environments and components. This interoperability is essential in reducing the effort that goes into the verification of an entire product. These methodologies, like OVM, UVM, define how bus functional models, or the BFMs, transactors *etc.* are designed to provide stimulus to the DUV, and defines standards for integration of these verification components into the verification environment. When put together in a coherent methodology, these designs provide a verification environment which greatly reduces both the time and effort required for verification.

OVM is explicitly simulation-oriented, OVM can also be used alongside assertion-based verification, hardware acceleration or emulation [7]. It is an open source methodology for functional verification which uses System Verilog from Mentor and Cadence. It combines mentor's AVM and Cadence's ERM methodology features. It is scalable to system level verification. An OVM testbench is composed of reusable verification environments called OVM verification components (OVCs). This improves productivity and reusability. The maintenance of verification components becomes very easy since they are standardized. It is not required on the part of the test developer to understand the entire architecture because it provides independence between the different components.

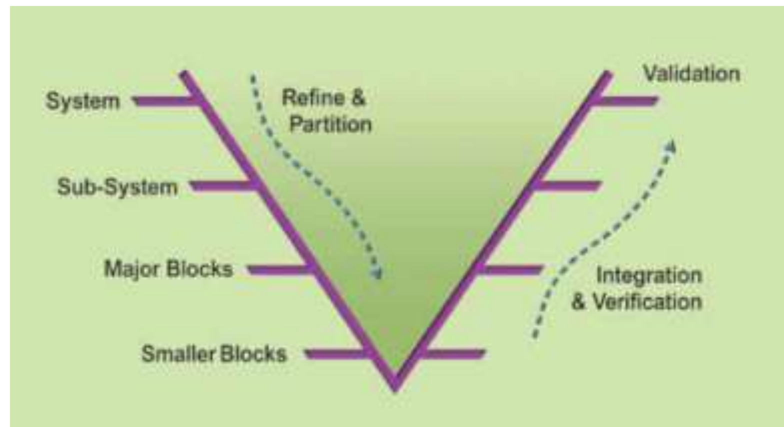


Figure 1.2 ASIC Verification Methodology [2]

The process of verification parallels the design creation process [8]. Usually, when faced with the task of verifying the design, first a directed approach is used. Test cases are written such that they exercise individual featured of the design. The stimulus for these test cases is written such that it verifies those features of the design. The test vectors are simulated and the waveforms and log files are then analyzed to confirm whether the design is behaving according to the expectations in terms of the functionality of a specific feature. This directed approach of verification can implement 100% coverage, but the time that goes into verifying each and every test feature with its set of test vectors takes a lot of time and resources.

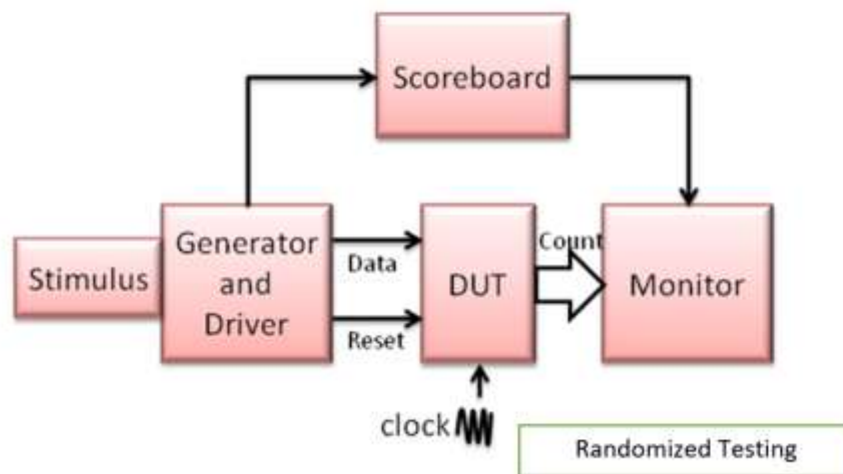


Figure1.3 Randomized Test Vector Verification

Several efforts have been made to automate the verification process, by the design specific nature of the tests and the unique response checking mechanism required for every individual design makes it very hard. True automation is expected to produce the test cases that are conventionally written

manually. However, a source for stimulus generation, if properly randomized can produce automatic random stimulus. One of the biggest reasons why randomization is very important in stimulus generation is that it produces stimuli that might have not been foreseen to be important. Randomization of input stimulus greatly increases the coverage and unfolds the corner points, where the design could actually fail in the real world. If the stimulus required by the design cannot be produced by an unbiased randomization, constraints can also be added to meet the stimulus requirements. Verification environment can also be reused across projects which increases its utility. An ideal verification environment would require just the reconfiguring of test cases specific to the design under test, with all other components remaining more or less the same. However, this modularity is very hard to achieve because of the complexity across designs and non-uniformity of the expectations from the verification of each individual design under test. Reuse is also a means to form a design specific verification environment. A reusable verification component should be able to meet all the requirements of the different blocks in a design and should also be able to verify the blocks at a system level.

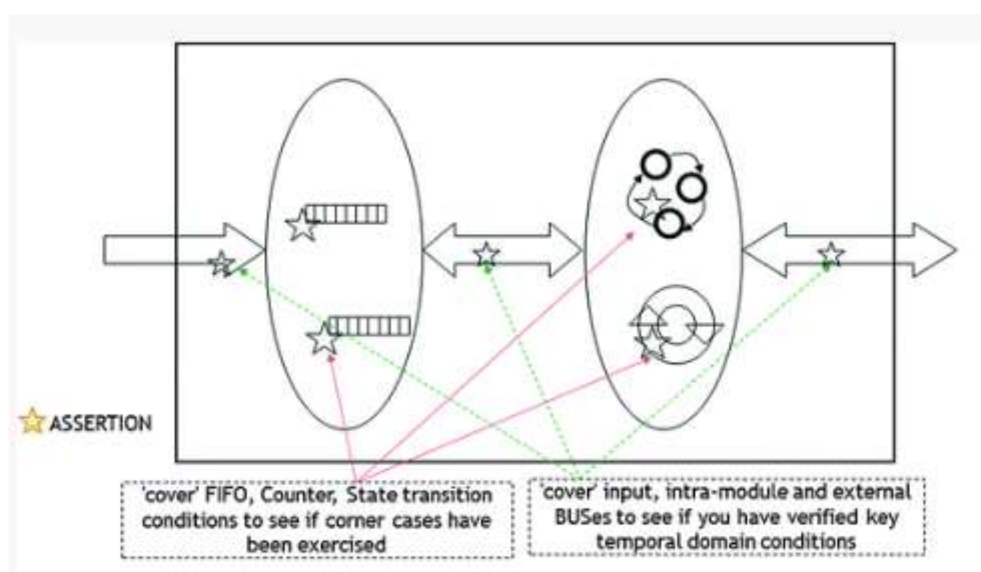


Figure1.4 Assertions provide temporal domain for functional coverage [9]

The three main sources of errors in post silicon, taped out chips are design implementation errors, errors in the specification of the design and the errors which creep in due to the reuse of modules as well as IPs. The latter error can be due to an internal error in the IP or the module itself, or can be because of improper usage. Most of the errors which escape the pre silicon process are due to unwritten or unaccounted for specifications which are not included in the test cases used to verify an IP and thus go unaccounted for.

When the RTL is designed, often assumptions are made for the surrounding designs and their behavior and impact on the design. These assumptions are sometimes not verified in simulation and hence go unverified. Any unexpected change in the behavior of the surrounding designs or a failure in synchronization of these designs may violate these assumptions, leading to failures. Assertions come very handy in such situations. An assertion is simple a check against the specifications in a design that is never expected to be violated [9]. If any violation of these specification occurs, they throw an error. The assumptions made in the design can be written in form of assumptions such that if these assertions fail, the exact point in time can be reached where the failure took place and hence the debug process becomes much easier. If these assumptions fail in the post silicon stage, then to root cause the exact issue which causes this failure becomes quite cumbersome. Another benefit of assertions is that they can be incorporated in all IPs and modules which are intended to be reused. These assertions can then specify the exact requirements for the reuse of a particular IP or module and hence the errors which creep into the design due to improper reuse can be avoided. Assertions can also be covered for functional coverage and thus provide valuable dual usage. They provide temporal domain functional coverage.

Debug begins with the execution of a detailed plan to characterize and validate both the functional and electrical operation of a design. This can be a daunting prospect with a complex design with requirements that operation needs to be checked over the entire voltage, temperature, frequency and process variation expected for the design, and that such a unit may be only one small piece of an entire design. The entire debug process consists of three main efforts: characterization, triage and debug. Characterization is the process of collecting data on many devices using as much validation content as possible. The validation content may include focused testing, where specific tests are written to exercise the device, or random testing, where random tests are generated to exercise the device. The goal of triage is to bucket failures into major common categories to allow the debug process to begin. With identified categories of failures to investigate, the process of debug can begin.

In the past, the validation of analog CMOS designs was possible with the help of schematic netlist simulation. As manufacturing processes grow in complexity, the ever-increasing amount of design-for-test/manufacturability/yield/quality (DFx) circuitry renders transistor-level simulation impractical; and digital behavioral simulation ignores crucial analog interactions [10]. This poses the requirement of an Analog and Mixed Signal (AMS) environment, which is robust on the basis of the functional coverage achieved by it. A top to down approach allows parallel development of schematics and behavioral functional models. On Intel's latest test chip this validation method discovered more bugs with less individual hours than past trials with absolutely digital or transistor-level validation.

One of the major problems seen in the pre-silicon validation realm today is the lack of a unified framework for all pre-silicon validation efforts including security validation [11]. Absence of a unified system for pre-silicon validation shows a communication challenge among various validation

groups. While every group keeps up and upgrades a specific methodology based on project supported, updating with changes and imparting the changes among other groups is to be done. Pre silicon verification refers to the validation that is done prior to the design being manufactured on silicon using simulation tools. It aims at verifying the correctness of the design with respect to the design specifications. It is critical in ensuring the quality of the product. The pre silicon verification done should be such that any alteration in one tool used for simulation does not affect the design by simulation a behavior that is different to that mentioned in the design specifications. Tools that are not synchronized with each other leads to a situation knows as the fragmentation problem. The tools used by different teams in the pre silicon validation flow are different and hence moving the design being tested across them to test different aspects of the design creates an ambiguity. The effect of such problem is that different tools end up simulation the same design specification in different ways. A unified framework for validating the functional correctness of a design is very important such that the fragmentation problem is eliminated in the pre silicon stage between the micro-architectures and complex architectures.

The growing importance of post-silicon validation in ensuring functional correctness of high-end designs increases the need for synergy between the pre-silicon verification and post-silicon validation [12]. There needs to be similarities in the verification methodologies in pre and post silicon environments. One such similarity could be the language in which the tests are written and the stimulus is specified. The test plan, the stimulus generation and the coverage metrics need to be in synchronization in a methodology that tries to integrate pre and post silicon validation. The test plan indicated the various featured that need to be validated in the DUV. Each feature has a set of input vectors and coverage metrics associated with it.

In order to reduce the cost of re-spins (both mask costs and the implementation time), it is essential to validate the implemented design and to identify the escaped bugs as soon as the first silicon is available [13]. Internal signals are not accessible on silicon, and this inability reduces the controllability and observability in the silicon. To be able to collect as much data possible from silicon so that debug is possible after manufacture, additional hardware has to be inserted in the design. Scan chains are one these additional hardware that is inserted in the Circuit Under Debug (CUD). Another method is making important signals, FSM states available to be observable for debug such that the error can be root caused for any bug observed in silicon.

One of the key components to enabling the debug process to go smoothly is the implementation of features in the design to aid debug. This is known as design for debug, or DFD. In addition, there are a number of tools and methods which are used in concert with these design features once silicon has arrived to accelerate the debug process. Typical post silicon methodology consists of booting low-level console user interface, running legacy tests, doing a boot of the favorite operating system, running pre silicon tests, running post silicon tests, locating and diagnosing bugs, reproducing and

analyzing bugs on the RTL model, and using microcode patches to provide a workaround so that validation may proceed. This is typically done with low observability using the small amount of DFT features available on the chip. Post silicon validation effort is done along two major directions: system validation and compatibility validation. System validation is primarily focused on validating the CPU and chip sets in an embedded systems environment that uses the new silicon in a multi way configuration and has special monitor software to download the validation tests into the platform for execution on bare silicon.

Pre-silicon alone cannot be relied upon to capture all bugs since simulation is several orders of magnitude slower that actual hardware. Popular pre-silicon verification techniques such as constrained random simulation and formal verification all suffer from scalability; therefore, they are infeasible for full-chip verification [14]. For post silicon verification, the internal signals are not readily available to be observed and so the debug process becomes all the more difficult. It is only in the post silicon stage that the actual environment in which the product has to operate is available and therefore can be tested in. The number of cycles for which simulation can be done in the pre silicon environment is very less and may miss out on corner cases. This makes post silicon verification even more important.

As the circuits are having an increasing rate of complexity, electrical bugs such as signal integrity can only be observed when the actual silicon is manufactured. Other electrical bugs such as crosstalk, thermal noise, power supply noise can also be accounted for only after chip manufacturing. What further increases the difficulty in post silicon debug is to identify whether a particular bug has arisen from a design error, or because of the physical factors that come into picture in silicon. With the help of automatic test pattern generation, scan chain insertion techniques, faults such as stuck-at-faults, transition faults, etc. can be detected. Most of these techniques require the design to be operating as a combinational design in test mode. This is also one of the challenges to be achieved for bug localization in validation at post silicon stages.

Internal Signal Trace Analysis (ISTA) is a model for post-silicon validation [15]. The goal of ISTA is to give the bug estimate, which includes the time the bug is detected in the post silicon environment. ISTA consists of an Error Detector, a few equipment tests called Signal Trace Probes (STPs) which are inserted in the chip, and an off-chip Instruction Set Simulator (ISS). Amid post-silicon acceptance, the data procured by STPs is investigated to recognize whether an error has happened and if happened, give the happening time and spot of the bug.

The High-Speed Serial Interface (HSSI) is an integral part of communication systems today [16]. An adaptable plan to quicken the post-silicon validation by utilizing a novel jitter infusion plan and a FPGA-based Bit Error Rate Tester (BERT) to test HSSI, without the need of Automatic Test Equipment (ATE) instruments has been developed.

Post-Silicon Validation faces numerous challenges in the areas of test generation efficiency, time utilization and comprehensive coverage of the various functionalities of advanced microprocessors [17]. The state space of modern processors is approaching infinite values, and the existing methodologies to generate tests are facing difficulties to cover even a small part of this state space, especially in an environment that is constrained in terms of time. Today's multiprocessor systems with their massive sizes have a lot of data sharing among them and this results in the need for collision checking, and hence, many combinations of outcomes of the data that is shared are required to be generated. This leads to a great need of intelligent test pattern generating systems which can effectively cover all the required test case scenarios. A methodology has been proposed that utilizes the idea of building a Master Test Program that is utilized to assemble numerous test-streams. It dynamically generates ample number of test cases using a Lightweight Runtime Regenerator. It greatly reduces the time required in the generation of test cases as compared to the conventional test cases generation methods.

Formal Property Verification tools based on abstraction refinement have also evolved. It iteratively refines an abstract model to better approximate the behavior of the original design in the hope that the abstract model alone will provide enough evidence to prove or disprove the property [18]. These tools initially started with designs consisting of not more than 500 registers. However, real world design have number of registers far greater than that. In further refinement, these tools are now able to verify design consisting of thousands of registers such that they find real world applications. ATPG techniques can also be used to find the error traces for the assertion properties that fail for these static checks. Refinement schemes can be identified when the error traces are found out for a given design. These algorithms identify the set of registers using three value simulation because of which the design can go bad, and not execute how it is expected to. Then, it applies the minimization algorithm to remove these potential set of registers. With continuous refinement and minimization, the design is found to be violating, or approving of a particular property that is being verified. A monolithic integration of SAT and BDD-based techniques could combine their individual strengths and result in a powerful solution for a wider range of applications [19]. Both these techniques have their distinctive strengths and weaknesses. They suffer from exponential worst case complexity, in which the elaborative nature of the design that is being verified renders the techniques into highly complex problems. High redundancies are one of the main reasons behind this complexity, in which a large part of the design contains redundancies in the generated netlists. The popular selling point of Formal Property Verification is that it is exhaustive in nature [20]. Although, this exhaustive nature might come off as a misnomer. Exhaustive property does not guarantee 100% coverage, it only means that the property that is being verified has been checked over all the possible behaviors of the design. This elucidates how important it is to write the verifying properties, and how the intent of coverage must be very well perceived beforehand for a particular design.

The design cycle of a product consists of addition and modification of many features in the design. These additions and modifications should not affect the existing design specifications. To ensure that any additional feature, or the modification of an existing feature does not affect the required specification of the existing software, regression testing is done. In the regression analysis, the tests which ensure proper functionality of the design are run whenever there are any changes in the design. Since design cycle involves modifications throughout the development phase, these tests need to be run periodically.

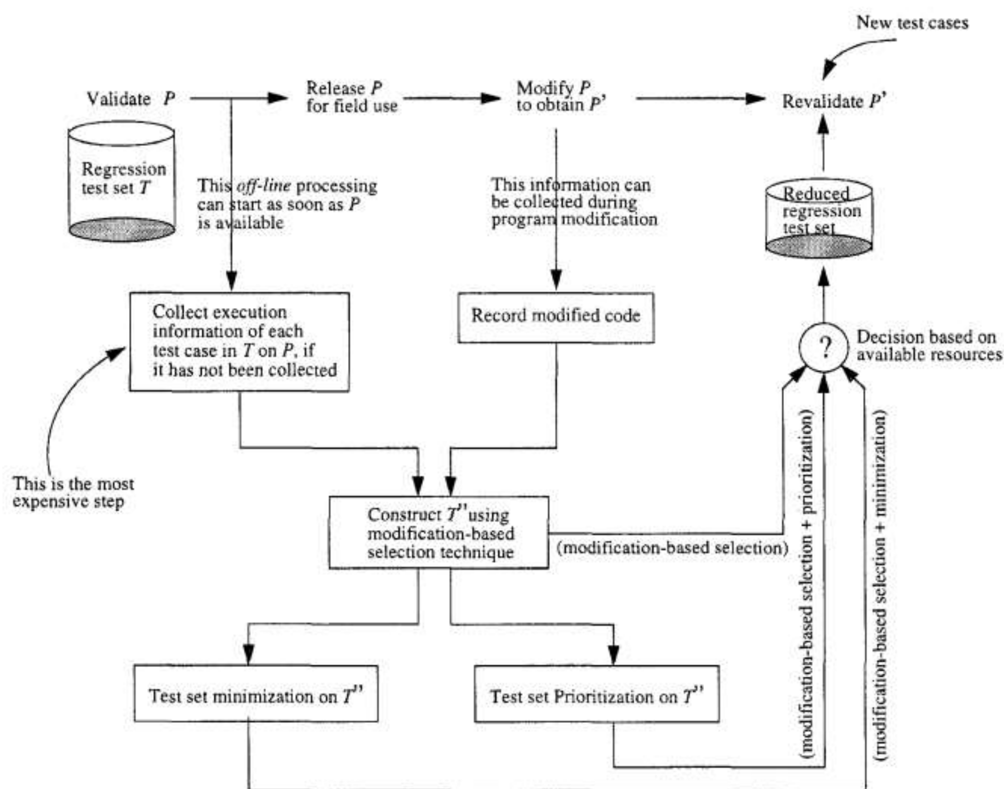


Figure1.5 Selecting effective regression test cases [22]

Regression testing is usually performed by running some, or all, of the test cases created to test modifications in previous versions of the code [22]. Regressions can be done with all the test cases, or a subset of cases can be formed which consists of all the specific features that need to be checked continuously throughout the design cycle, since they are most likely to get affected.

The selection of these effective test cases is done with the help of various algorithms. Regressions with the entire set of test cases can be both costly, as well as time consuming. However, cutting down the regression tests should not compromise the coverage offered by these tests on the expected features of the design. So, the selection of these mini test lists is very crucial. One of the methods is modification based test selection [22]. This is done by re-running only those test cases from the entire

set, in which the recently modified and the previous design produce different results. Minimization and Prioritization are some other techniques for automatized minimization of tests run on the design.

CHAPTER 3

VERIFICATION METHODOLOGY

3.1 OPEN VERIFICATION METHODOLOGY

Open Verification Methodology (OVM), describes a way to organize and build verification infrastructures. The overarching principle that motivates this methodology is not a new one, but it is infinitely practical – keep it simple. The objective of this methodology is to design testbench architectures that are easy to build, debug, maintain, and understand. This verification methodology was chosen because it meets the following verification requirements:

- Reusable across abstraction layers and design (parameterization, generalization, minimize dependencies, well defined semantics)
- Modular (localization of functionality, localization of data, communication through well-defined interfaces)
- Use standard interfaces (provide external view of object, hide implementation details, define interface semantics)
- Support stepwise refinement and abstraction (keep things at the highest level possible)
- Implemented in System Verilog and currently supported by VCSMX (2009.06+)
- First truly open, interoperable, and proven verification reuse methodology
- Most advanced methodology, enabling multi-language plug-and-play VIP
- Integrated with the proven Incisive Plan-to-Closure Methodology

The OVM provides a library of base classes that allow users to create modular, reusable verification environments in which components talk to each other via standard transaction-level modeling (TLM) interfaces. It also enables intra- and inter-company reuse through a common methodology and classes for virtual sequences and block-to-system reuse. The reuse concepts within the OVM were derived mainly from the URM (Universal Reuse Methodology) which was, to a large part, based on the eRM (e Reuse Methodology) for the e Verification Language developed by Verisity Design in 2001. The OVM also brings in concepts from the Advanced Verification Methodology (AVM).

Every interface has its own agent and Inside every agent, it possess a driver item, which drives sequence item to the interface which forms connection between the design under test & the agent and monitor. The monitor represents the pin stage activities of interface, and after that it does the conversion of the activities into the transaction items, and transmits the transaction item at higher level OVM components through OVM port i.e. analysis port and through export.

The components used in Open Verification Methodology have been defined below. The interaction between the testbench components as well the Design Under Test (DUT) has been specified.

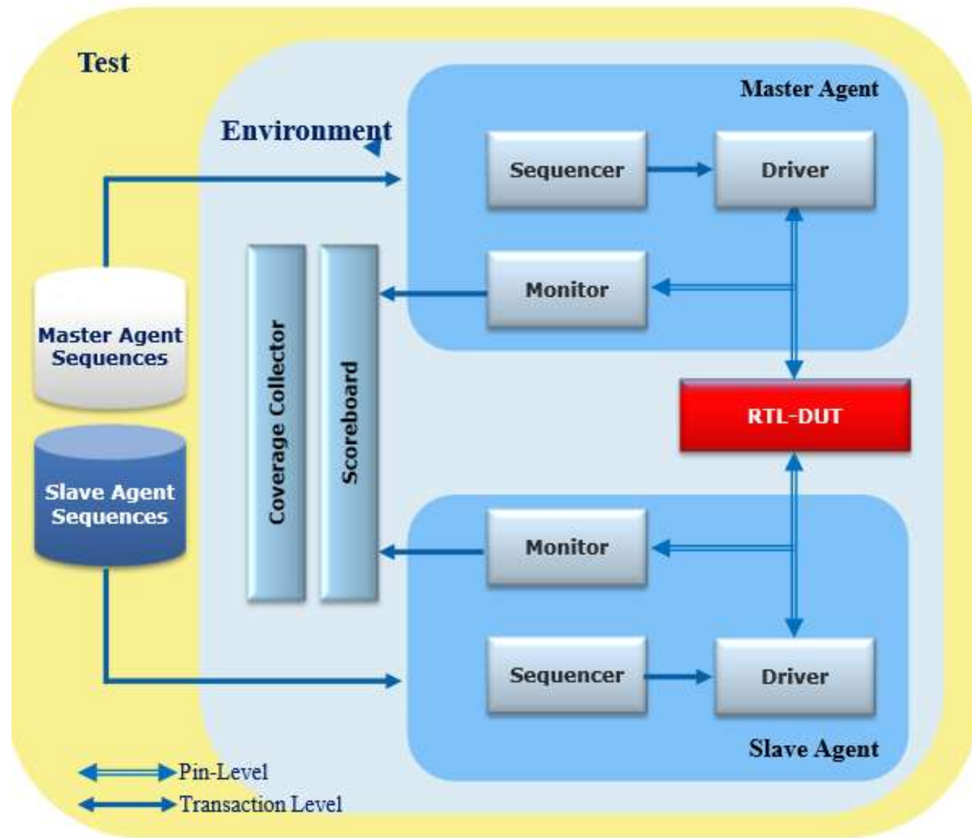


Figure 3.1 Open Verification Methodology [2]

1. Design Under Test: It is the Design Under Test. It is a pin level component whose design is being verified by the testbench. The first task in building the testbench environment is to connect the DUT to the class based OVM testbench in the top level module.

2. Agent: The agent further consists of the following components:

- **Sequencer:** A sequencer routes a sequence to the driver. A sequence is a series of transactions. User can define the complex stimulus. Sequences can be reused, extended, randomized, and combined sequentially and hierarchically in various ways. It routes the transactions carried out in the test.
- **Driver:** Driver converts a transaction level stimulus to pin level activations on the DUT. In the run_phase(), the drives synchronizes with the clocks present inside the DUT and with the sequencer which routes the stimulus to the driver.
- **Monitor:** A monitor observes the pin level activations and then converts them into transactions. It does the reverse of driver. It is passive in nature as it does not drive any pins.

3. Scoreboard: A scoreboard tracks transaction level activity that is going in and out of the DUT to ensure that DUT is functioning properly. For example, it may track packets in vs. packets out to see if all the packets sent into a communication device made it out intact.

4. Coverage collector: A coverage collector has counters organized in bins. It can tap into various observable points in the test-bench. It simply counts the transactions that are sent to it and puts the counts in the appropriate bins.

5. Env: It instantiates all the agents, scoreboards, coverage collectors and connects them. It also contains the default configuration of the test-bench, standard end of test routine, initialization routine etc.

6. Test – The test instantiates the Env. It can re-configure the Env based on test intent. It controls running of sequences on the available agents, can over-write transaction constraints etc. The test must be first registered with the factory and must be able to implement the various phases, like the build_phase() and the run_phase(). The build_phase() inside the test is used to build and then instantiate the test environment. In the run_phase(), the execution of the test case is carried out. An OVM test has several basic responsibilities:

- Get the virtual interface handles from the configuration database
- Instantiate the OVM environment
- Use the configuration database to pass the virtual interface handles and other information down to the environment
- Instantiate and start any sequences necessary for the test case
- Manage phase objections, to ensure the test successfully completes

OVM enables the creation of modular as well as reusable verification components, tests and environments.

In directed test case methodology, test stimulus are generated in the test case. The test stimulus contains test case files and register programming files. For ex. For a multiplication, the test case stimulus will include three numbers number A, number B, and number C golden reference files. Actually directed test cases are the basic level of verification. These types of test case are particularly dedicated for one configuration. The test stimulus generation is quite easy in this type of test cases. The advantages of directed test cases are as follows:

- Basic entry level verification
- The complexity of the design at this level is not that much high.

By using system Verilog language we can excellently verify the micro architecture of the design. The language provides various constructs especially for verifying the design. The system Verilog standards also have a separate class for randomization.

3.2 SYSTEM VERILOG ASSERTIONS

Assertions are primarily used for the validation of a design. They can basically be considered as active comments that specify the legal or the illegal behavior of the design under verification. They can also be used to specify whether the test that is being used to verify the functionality of the design is having proper coverage or not. Assertions can be used to dynamically check the behavior of a design in a simulation environment. Assertions are incorporated in the testbench and then are evaluated to be either passing or failing depending upon the condition that they are checking in the design. Assertions can also be used statically, with the help of a formal verification tool which is essentially a property tool checker, which proves whether the design is meeting its specifications or not. Dynamic property checker works on the basis of certain assumptions that need to be provided about the behavior of the design.

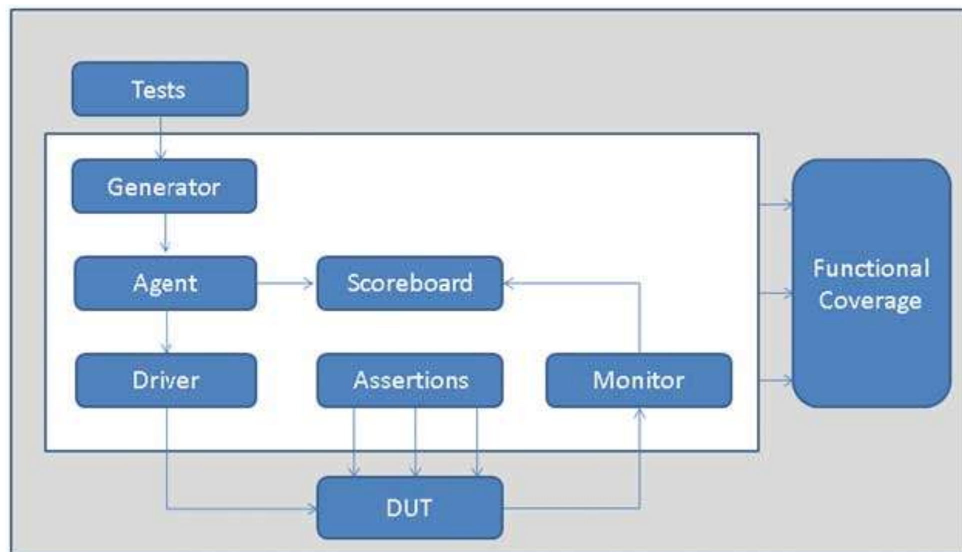


Figure 3.2 System Verilog Assertions in Test Environment

To achieve complete SoC or Silicon Realization flow, assertion based verification forms one of the key contributing factors. Assertions help to capture the design intent in such a manner that it is executable and concise. They act as monitors during the simulation phase and report whatever errors that they find close to their source, and then also provide the coverage information.

Assertions have various advantages over standard Verilog or System Verilog codes. They are easier to write, thereby increasing the productivity of the design, they are faster to debug since any violation throws an error specifying which property of the design is not behaving as expected, and also provide the information on functional coverage.

There are two types of System Verilog Assertions, Immediate and Concurrent.

- **Immediate Assertions**: These are simple procedural statements that check whether the condition that they are specifying is true or not. They are similar to the 'if statement', except that if statement only checks the condition, whereas immediate assertions assert that the expression is true.

  if (A==B) // Simply checks whether the value of A is equal to the value of B
  assert (A==B) // Compares the values of A and B and throws an error if they are unequal

  An immediate assertion asserts a conditional expression. If this conditional expression evaluates to be 0, Z or X, then the simulator throws an error. An immediate assertion can also include a statement that indicates whether the assertion has passed or failed.

  assert(A==B) $display ("Values are equal")

  else $display ("Values are unequal")

  Immediate assertions can be added in a simulation to throw errors whenever an expected signal value assignment goes wrong.

- **Concurrent assertions:** These are used to validate a property. These properties can also specify the timing information that details when a particular condition has to be examined. These assertions have the following syntax:

  assert property (@(posedge Clock) Req |-> ##[T] Ack);

  An expression used in an assertion is always tied to a clock definition. The sampled values are used to evaluate value change expressions or boolean subexpressions that are required
  to determine a match of a sequence. The keyword property distinguishes a concurrent assertion from an immediate assertion.

  Specific clock instances, at which a property needs to be validated can be added. The property is then checked at these clock edges only. 'Req' represents as simple Boolean expression, which when comes true, the other side of the assertion is evaluated. '|->' is the *implication operator* which indicates that whenever req is true, the expression following it is evaluated. ##[T] represents the time interval, or the clock cycles after which the evaluation takes place. The implication operator allows the monitoring of the precondition and evaluates what follows it only after the precondition is satisfied. The expression at the left hand side of the implication operator is known as the *antecedent sequence* and the right hand side of the implication operator is known as the *consequent sequence* expression. Implication operator is of further two types, overlapped and non-overlapped. For overlapped implication operator, '|->', when the antecedent sequence evaluates to be true, the elements on the consequent

sequence are evaluated on the same clock cycle. For the non-overlapped implication operator, ‘|=>’, when the antecedent sequence evaluates to be true, the elements on the consequent sequence are evaluated on the next clock cycle. The hierarchy of the assertion layer is given below:

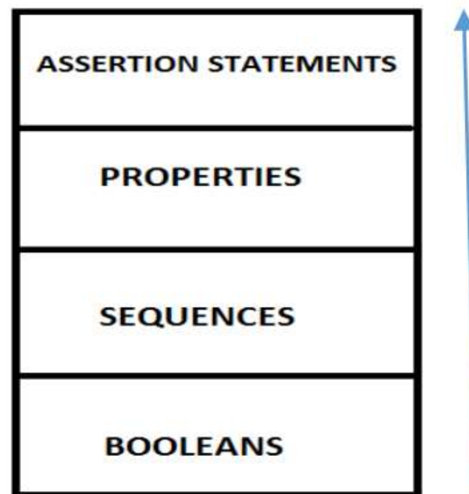


Figure 3.3 Hierarchy of Assertion Layer

- **Boolean layer:** Boolean layer is the lowest layer in concurrent layer, where Boolean expression checking is done on variables. The result of Boolean checking is either true or false. If the variables contain x or z, then the result is false.

  Example

  (en && ce && addr < 100)

- **Sequences layer:** Sequence Layer uses the Boolean layer to construct valid sequence of events. The simplest sequential behaviors are linear. A linear sequence is a finite list of System Verilog Boolean expressions in a linear order of increasing time

  Example

  Sequence req_gnt_0clock_seq;

  req[2] ##0 gnt[2];

  endsequence14

- **Properties layer:** Property layer is similar to sequence layer only difference is it implication operator. It means when left of operator is true then right operator has to be passed.

Example

```
property req_gnt_0clock_prop;
@ (posedgeclk)
  disableiff (reset)
req[2] |-> req_gnt_0clock_seq;
endproperty
```

- **Assertion layer:** Assertion layer will assert all properties defined. Along with assertions assume and cover can also be used.

  Example:

  ```
  req_gnt_0clock_assert : assert property (req_gnt_0clock_prop);
  ```

Assertions can be directly used inside the procedural source code where design specification are specified, they can be included in a separate program, which is further bound to a specific module. Assertions shorten the time required to check a functionality.

Consider a property that has to be validated that states that whenever grant is de-asserted, which means it goes high, the request signal, req should be asserted, which means it should go low. This property checks some crucial requirements of an interface. Assertions are concurrent, as well as multi-threaded.

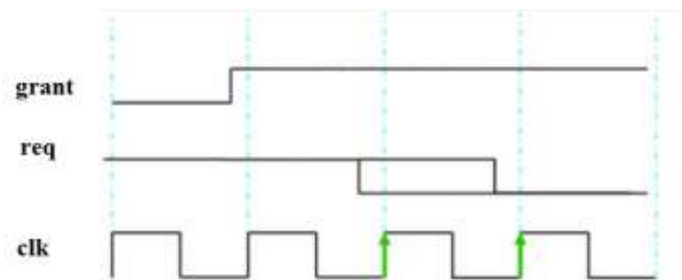


Figure 3.4 Property to be validated

There are many ways to code the required checker for this property. One of them is illustrated in Figure 3.5.

```
1
2 // Verilog Code
3 always @(posedge grant)
4 begin: rg_check
5  @(posedge clk);
6  fork
7   begin
8    @(negedge req) disable rg_check;
9   end
10  begin
11   repeat(2) @(posedge clk); $display ("rg_check FAIL");
12   disable rg_check;
13  end
14 join
15 end
16
```

Figure 3.5 Verilog code to verify the property

Two procedural blocks have been forked out, one of which monitors req, and the other waits for two clock cycles. The entire block, rq_check can be disabled when any of the two procedural blocks get completed. The code is not very easy to interpret and contains more information on how the checker has to be implemented, than what it is trying to check. On the contrary, the SV assertion code is given in figure 3.

```
17 // Assertion
18 property rg_check;
19 @(posedge clk)
20 $rose (grant) |-> ##[1:2] $fell (req);
21 endproperty
22 assert property (rg_check) else $display("rg_check FAIL");
23 cover property (rg_check) $display("rg_check PASS");
24
```

Figure 3.6 SV Assertion to Verify the Property

The code is quite simple and exactly states what is being checked at what instant. The assertion can be debugged easily, as it will throw an error only at the time the property fails. The Verilog code shown earlier, however, is not only hard to interpret, also is quite prone to errors.

Assertions also improve the overall observability of the design. They are located at the temporal region, which means that whenever they throw any error, they point to the main source of the bug, which is explicitly written in the form of the property. Contrary to this, whenever a test fails which is written using conventional System Verilog code, the error has to be back traced all the way to the design to debug it.

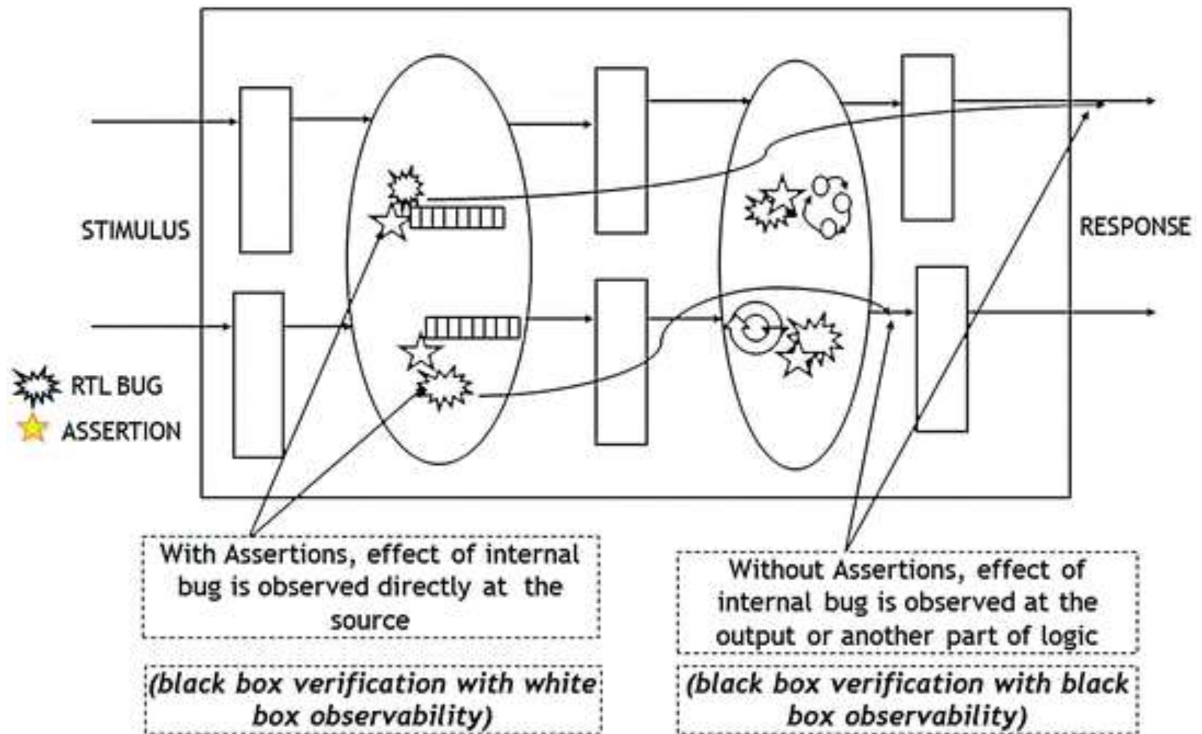


Figure 3.7 Improved Observability in Assertions [9]

Traditional verification techniques using conventional System Verilog and OVM methodologies can be termed as Black Box verification, since in it, inputs are applied to the design without caring about what is inside the box, and the outputs are also observed outside the box, with blackbox observability. Assertions allow us to do a black box verification with whitebox observability.

Assertions provide functional coverage. Absence of any error does not guarantee a bug-free design. The assertion shown above will only pass if the cover is hit, that is, if the antecedent expression is satisfied. This implies that assertions not only check whether a property is true or not, but also check whether the conditions on which a particular property is being checked are met or not. This ensures that the assertion won't pass in the absence of a violation, if the antecedent expression is never met.

Assertions are synthesizable and hence can be used to fire directly in the hardware. Synthesizable assertions get partitioned to the emulation hardware. During the time the emulation takes place on the hardware, if there is a bug in the design of the logic, the synthesized assertion will fire and as a result will trigger a stop/trace register which will further stop emulation and then it will directly point to the cause of failure.

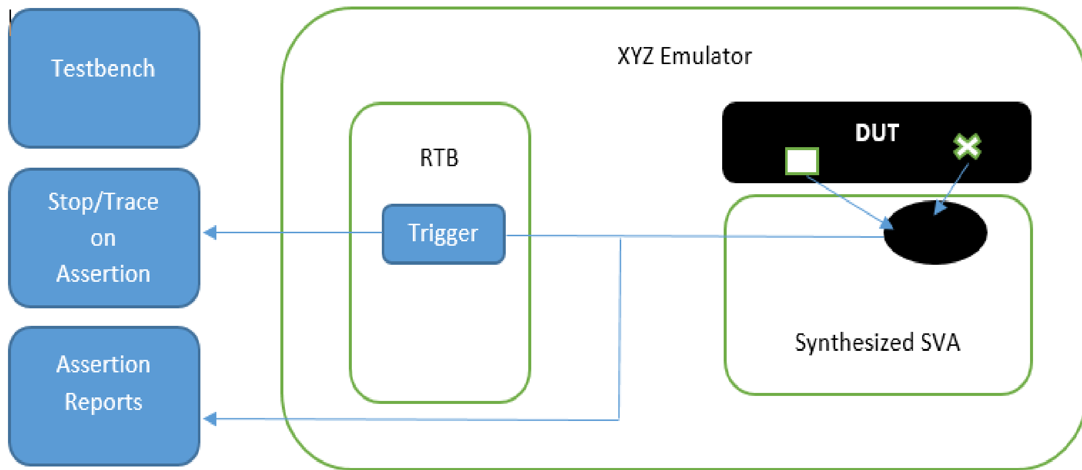


Figure 3.8 Assertions for Hardware Emulation

In figure 3.8, synthesized assertions are supported by XYZ emulator. The embedded and bound assertions are automatically synthesized and mapped into the emulator. Assertion reporting as well as triggering happens at run-time, which is similar to simulations. They also give added compile time and run time control over assertion triggering and reporting.

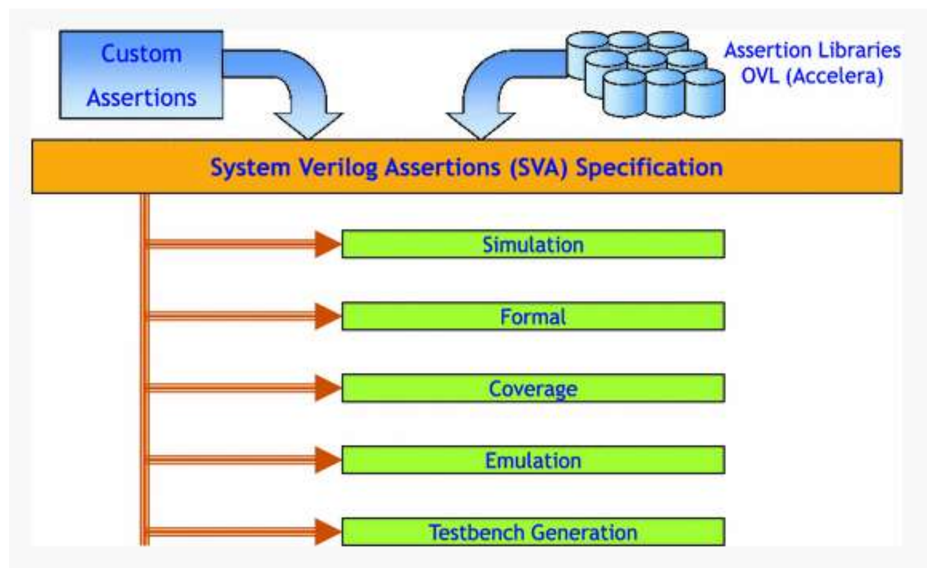


Figure 3.9 Assertions and Open Verification Library for different uses

Assertions can also be synthesized in silicon. During the process of post silicon validation, if there is any functional bug present, an assertion can be fired and the failure can be recorded in a hardware register. This register can be monitored via the GPIO of the chip, or can be scanned out using the

JTAG boundary scan. The corresponding output can be decoded to determine which assertion got fired, and hence which part of the silicon caused the resulting failure.

Today, assertion-based verification (ABV) has been successfully applied at multiple levels of design and verification abstraction—ranging from high-level assertions within transaction-level test benches down to implementation level assertions synthesized into emulation and hardware. With the advent of standardized assertion languages and assertion libraries, the industry has recently witnessed an increased interest in adopting assertion-based techniques.

3.3 FORMAL VERIFICATION AND SIMULATION

Formal Verification (FV) has emerged as a method in ASIC design to reduce the time to verify a design, not by accelerating simulations but by eliminating them once the initial design has been functionally validated. FV is a mathematical method used to directly compare the logical function of one design with that of another. It builds mathematical models, i.e., equations of the logic, and tries to prove that the equations derived from both the specification and implemented designs are equal unlike simulations, which rely on the user to specify the portion of the design to be tested by writing and executing test cases. The analytical approach used in FV is inherently complete. Formal verification is basically a methodology which proves the correctness of a system with respect to certain formal specifications. Formal Verification algorithmically and exhaustively explores all possible input values over time. Currently the FV effort is not to replace simulation-based verification, instead, it is to augment pre-silicon validation by proving the total correctness of high-risk functionality. Design simulation would catch bugs, but FV is used to ensure that no corner cases are missed. Formal property verification is one, where CAD tools are used to mathematically prove that for a given piece of RTL (usually written in Verilog), a set of properties will always be true. It is a static analysis for all possible inputs, making it a very powerful technique that is qualitatively different from methods such as simulation which just check particular test. Formal Property Verification, or FPV, is a powerful technique in which properties of a VLSI design are proven mathematically. This contrasts with traditional simulation-based validation, which only applies specific test vectors— FPV proofs are valid for all possible test vectors, theoretically providing 100% coverage for the properties verified. To quantify the improvements due to FPV, in terms of the design being closer to given specifications, we can make use of code coverage and functional coverage. Functional coverage observes execution of a test plan. As such, it is code written to track whether important values, sets of values, or sequences of values that correspond to design requirements have been exercised. Functional coverage is important to any verification approach since it is one of the factors used to determine when testing is completed. Specifically, 100% functional coverage indicates that all items in the test plan have been tested. Combine this with 100% code coverage

and it indicates that testing has been completed. We may also include the power intent for the design using unified power format (UPF).

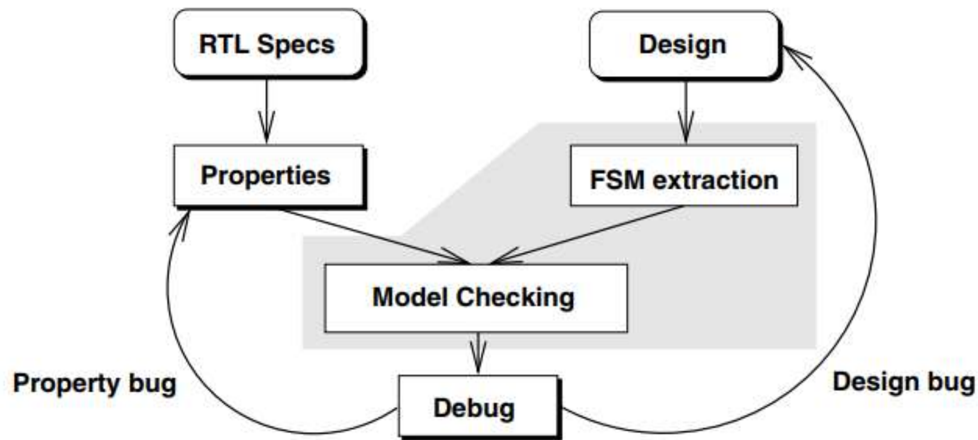


Figure 3.10 RTL Formal Property Verification [20]

Another selling point for formal verification is completeness, in the sense that it does not miss any point in the input space—a problem from which simulation-based verification suffers. However, this strength of formal verification sometimes leads to the misconception that once a design is verified formally, the design is 100% free of bugs. As opposed to working at the point level, formal verification works at the property level. Given a property, formal verification exhaustively searches all possible input and state conditions for failures. If viewed from the perspective of output, simulation-based verification checks one output point at a time; formal verification checks a group of output points at a time (a group of output points makes up a property).

Mathematical models give a sense of reliability in terms of the functional coverage that is achieved by them. The entire design is taken into consideration, and all the possible trajectories are analyzed. Comparison between conventional verification techniques using simulation and formal verification can be done on the basis of several factors.

Table 3.1 Comparison between FV and Conventional Verification

SNo.	Parameter	Simulation	Formal Verification
1.	Inputs	Testbench drives the values on the inputs, un-driven values are taken to be "X"	Free inputs take every possible value, in every possible combination and in every possible sequence.
2.	Testbench	Required	Not Required
3.	Approach	Directed Test Vectors	Exhaustive
4.	Corner case coverage	Not guaranteed	Guaranteed
5.	Trajectories in State Space	Analyzes only one trajectory at a time	Analyzes all the possible trajectories in parallel
6.	Constraints	Testbench code which is non-synthesizable	By specifying assumptions, restricted input scenario can be avoided
7.	Consideration	Takes entire design into consideration	Only relevant part of the design
8.	Verification Time	Significantly high	Comparatively small

Formal verification gives many advantages over the conventional verification methods, though its usage must be carefully adopted. Since the formal verification is done with the help of mathematical modal tools, the inputs are given all the possible values that they can take by the tool itself. The inputs are thoroughly inspected by the tool deep through the design and all its dependencies are evaluated. The drivers of the inputs are then inspected to find out further dependencies from the design and all the possible combinations and evaluated to check whether a property can fail under any given scenario. This greatly improves coverage as the design is deeply investigated for underlying failure conditions.

CHAPTER 4

REGISTER SPACE ACCESS

4.1 INTRODUCTION

Registers and memory segments constitute a major part of today's complex IP's. Register information for most of the designs begins with a logic architect giving the register description. Once the description is prepared the hardware group, firmware group, and verification group can start coding particular viewpoints of the registers given in the functional specification. Starting from design to validation to documentation of each and every register, each bit and its property requires huge amount of time and is also tedious. Common Register Interface (CRI) is a protocol used to standardize common functions such as register access which promotes ease of integration and therefore expedites the time to market. The intent of a CRI compliant system should be as a bridge between the global requesters (such as JTAG) and the local configuration register space. The Register spreadsheet conventionally written in Microsoft Excel comprises of lane addresses, Data Word (DW) i.e. address of a register, default values, description etc. The address, data, read/write signals is passed to registers through JTAG BFM. The JTAG BFM converts this high level information to corresponding Test Data in (TDI), test clock signal (TCK), test mode select (TMS), and Test reset (TRST) and expects a correct Test Data Output (TDO).

4.2 COMMON REGISTER INTERFACE

Advanced Peripheral Bus (APB) is a register access protocol found in the Advanced Microcontroller Bus Architecture (AMBA) specification used within the ARM ecosystem. The primary short coming of APB is lack of support for byte enables. This limits a system to DWD width writes adding latency and complexity as a Read-Modify Write is required to change less than a DWD of data. CRI also offers the advantage of transmitting read and write data a single byte at a time. Since CRI is designed for a single master multi-slave architecture, decreasing the wiring congestion can lead to more compact implementations. CRI offers built in timeout counters to avoid hanging the system in the event that an endpoint does not respond. This increases the testability of CRI versus APB.

CRI protocol gives the way in which the internal configuration register space is to be accessed. The needs to be proper communication between the CRI master and the CRI slave. CRI master asserts the control signal to communicate to the slave that a request has been put by a global requester to access the common register space. The request can be either to read a particular register, or write it.

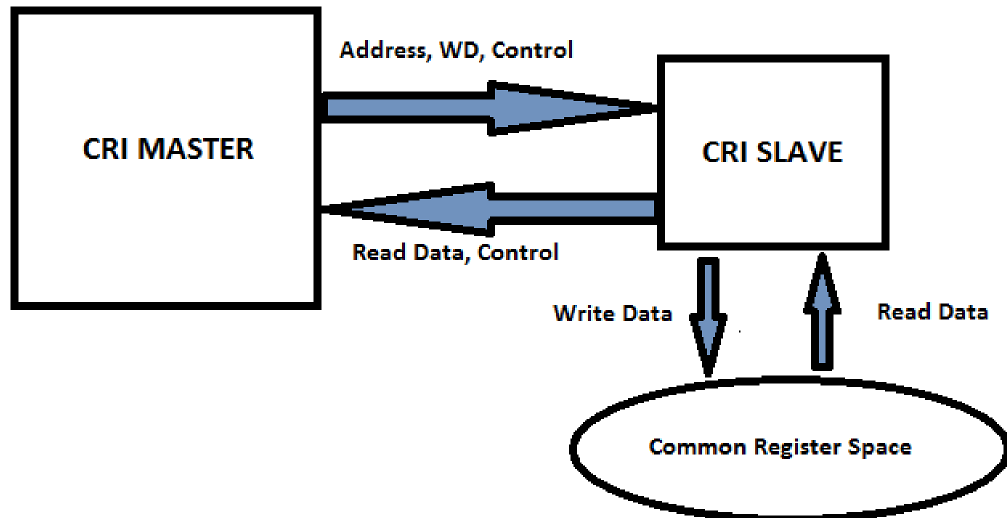


Figure 4.1 Register Space Access

4.3 JTAG PROTOCOL

4.3.1 Introduction to JTAG

JTAG (**J**oint **T**est **A**ction **G**roup) typically refers to the IEEE 1149.1 standard developed by the IEEE to define universal test access hardware architecture. It is generally used to test board-level connectivity and to access internal test registers used for enabling test hooks and observing internal signals.

JTAG implements standards for on-chip instrumentation in electronic design automation (EDA) as a complementary tool to digital simulation. It specifies the use of a dedicated debug port implementing a serial communications interface for low-overhead access without requiring direct external access to the system address and data buses.

The interface connects to an on-chip test access port (TAP) that implements a stateful protocol to access a set of test registers that present chip logic levels and device capabilities of various parts. The protocol enables to check the board level connectivity of the various TAPs present. These are connected to each other as shown in figure 4.2.

The entire system runs on the same clock and the individual TAPs are interconnected and interact with the common clock. These are various test devices that are checked for validating their interconnectivity. The protocol works on serial shifting of data, one bit per one clock into the test device and similarly shifting out one bit per clock to read out the data.

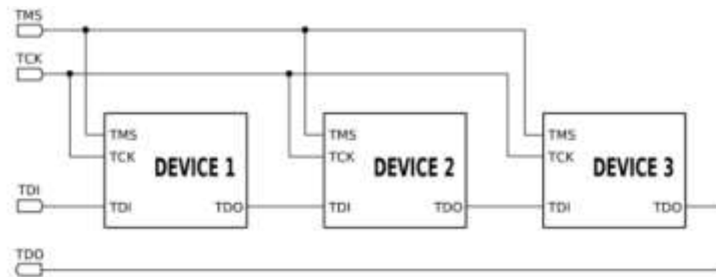


Figure 4.2 Interconnection of Various Test Devices

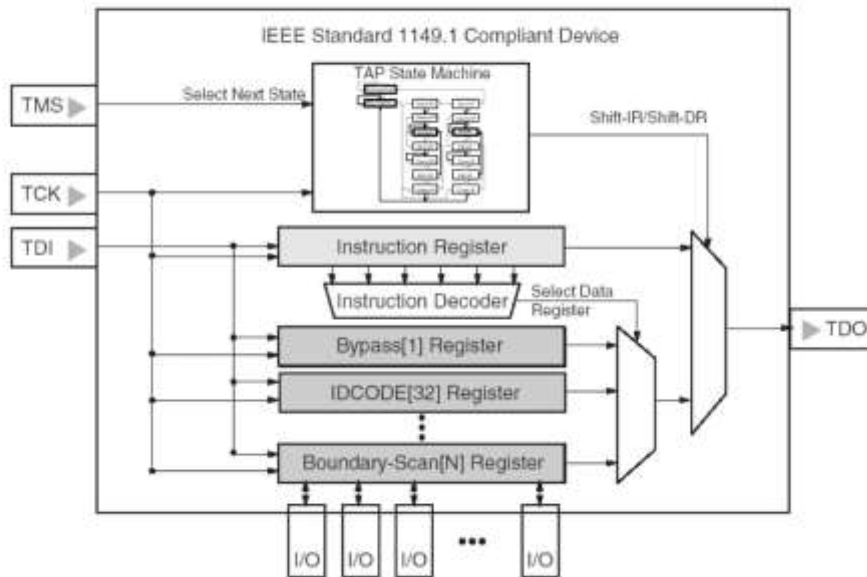


Figure 4.3 TAP Architecture

Figure 4.3 shows how the TAP and boundary-scan register are interconnected to form the overall product JTAG architecture. The TAP block is made up of a TAP controller, decode and control logic, and a number of registers (including the Boundary-scan Register). It is connected to the 5 JTAG pins: TCLK, TRST, TMS, TDI and TDO through a Master TAP (placed in the PHY). The TMS pin is used to maneuver the TAP controller to its different states; it is set to Vcc by default using a weak pull-up. The TDI and TDO pins are used to shift serial data in and receive serial data out respectively.

4.3.2 TAP State Machine

The TAP controller is a simple 16-state FSM. State transitions happen on the rising edge of Tclk and the next state only depends on the TMS signal. Since the TMS signal is pulled high by default, the TAP controller stays in the ‘Test-Logic Reset’ state until TMS is pulled low. In this state, the test

hardware is reset and disabled so that it will not interfere with the normal functionality of the design. When TMS is pulled low, the controller transitions to the 'Run-Test-Idle' state and stays there until TMS is set high again. In this state, any previously activated internal test is allowed to run; else, the test hardware is simply in idle. From the 'Run-Test-Idle' state, the flow splits into two main branches, each with 7 states. The branch closer to the 'Run-test-idle' state is taken if the user desires to access a Test Data register while the other branch is taken to access the Instruction register. But aside from which kind of register to access, each of the 7 states in the two branches performs the same function to their respective register type

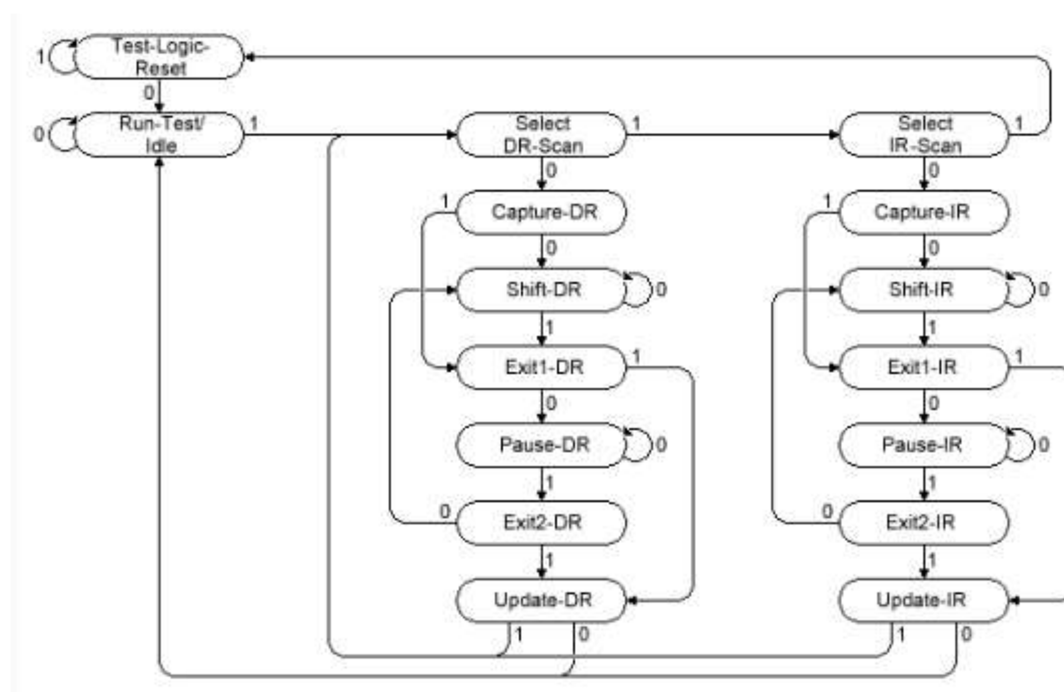


Figure 4.4 JTAG Finite State Machine

The little numbers ("0" or "1") close to each arrow are the value of TMS to change state. So for example, if a TAP controller is at state "Select DR-Scan" and TMS is "0" and TCK toggles, the state changes to "Capture-DR".

The following list describes the behavior of each state in the TAP:

- **Test-Logic-Reset:** In this state, the test logic is disabled so that normal operation of the device can continue unhindered. The instruction in the Instruction Register is forced to IDCODE. The controller is guaranteed to enter Test- Logic-Reset when the TMS input is held active for at least five clocks. The controller also enters this state immediately when TRST_N

is asserted active. The TAP controller cannot leave this state as long as TRST_N is held active.

- **Run-Test/Idle:** The TAP idle state. All test registers retain their previous values.
- **Select IR-SCAN:** Temporary controller states in which all test data registers selected by the current instruction retain their current state. Initiates register scan sequence.
- **Capture-IR:** In this state, the shift register contained in the Instruction Register loads a fixed value (irshiftreg[msb:2] = 0, irshiftreg[1:0] = 2'b01, the two least significant bits are "01") on the rising edge of TCK. The parallel, latched output of the Instruction Register ("current instruction") does not change.
- **Shift-IR:** The shift register is part of the overall Instruction Register that is connected between TDI and TDO and is shifted one stage toward its output on each rising edge of TCK. The output arrives at TDO on the falling edge of TCK. The current instruction in the Instruction Register does not change.
- **Exit1-IR:** These are temporary controller states. If TMS = 1, then on the next rising TCK, the state machine will enter the Update-DR/IR states.
- **Pause-IR:** Allows shifting of the Instruction Register (shift register portion) to be temporarily halted. The current instruction does not change.
- **Exit2-IR:** Temporary controller states allowing either resumption of or termination of the current scan instruction.
- **Update-IR:** The instruction which has been shifted into the IR shift register is captured into the Instruction Register on the falling edge of TCK. Once the new instruction has been latched, it remains the current instruction until the next Update-IR (or until the TAP controller state machine is reset).
- **Select DR-SCAN:** Temporary controller states in which all test data registers selected by the current instruction retain their current state. Initiates register scan sequence.
- **Capture-DR:** In this state, the Data Register selected by the current instruction may capture data at its parallel inputs.
- **Shift-DR:** The Data Register connected between TDI and TDO as a result of selection by the current instruction is shifted one stage toward its serial output on each rising edge of TCK. The output arrives at TDO on the falling edge of TCK. The parallel, latched output of the selected Data Register does not change while new data is being shifted in.
- **Exit1-DR:** These are temporary controller states. If TMS = 1, then on the next rising TCK, the state machine will enter the Update-DR/IR states.
- **Pause-DR:** Allows shifting of the selected Data Register to be temporarily halted without stopping TCK. All registers retain their previous values.
- **Exit2-DR:** Temporary controller states allowing either resumption of or termination of the current scan instruction.

- **Update-DR:** Data from the shift register path is loaded into the latched parallel outputs of the selected Data Register (if applicable) on the falling edge of TCK. This and Test-Logic-Reset are the only controller states in which the latched paralleled outputs of a data register can change.

4.3.3 JTAG Registers

TAP architecture has multiple data registers and only one instruction register. Instruction Register selects the type of data register to be selected depending on the instruction serially fed into it. Instruction register comprises of hold and shift segments. Hold segment holds the current instruction and shift segment is present between test data in and test data out pins. The control signals to the Instruction register begin from the TAP controller which results in shift in or shift-out through the shift segment of Instruction register (IR) or the instruction can be held in hold section, called update operation. The IR must be no less than two-bits in length in order to permit four instructions namely bypass, sample, preload and EXTEST.

1. **Data Registers** is a combination of three data registers available namely the Boundary Scan Register (BSR), bypass register and IDCODE register.
   a) **Boundary Scan Register** is the most important data register whose functionality is to move data from input to output pins of a design through the boundary scan cells.
   b) **Bypass Register** is a one bit register that transfers data from the input to output pin of a device.
   c) **IDCODE Register** holds the ID and revision number of the device thereby allowing the device to be connected to its BSDL file i.e. Boundary Scan Description Language file.
2. **Instruction Register** is a minimum two bit register that holds opcode of boundary scan tests to be performed.

4.3.4 JTAG Instructions

The four compulsory instructions in Institute of Electrical and Electronics Engineers (IEEE) 1149.1 are as follows:

1. **BYPASS:** Bypass register is used to capture, shift and update data from TDI to TDO pins of a device. This will permit the normal functioning of the chip.
2. **SAMPLE:** Boundary scan register is used to capture and shift data from TDI to TD0 pins of the device. Here the update stage does not drive data into input or output.
3. **PRELOAD:** Preload instruction causes values to be loaded into BSCAN cells and boundary scan register shifts the loaded data. In this case the capture stage will not capture the previous value into the BSAN cell and the update phase will not drive data into input or output. This instruction is usually clubbed with SAMPLE instruction.

4. **EXTEST:** The chip is put in extest mode before information is caught, moved and updated through the BSACN register. This is utilized to check connectivity between various chips. In extest mode the chip does not attempt to drive outputs or acknowledge inputs. It is ordinary to utilize PRELOAD to define up the BSCAN register before EXTEST.
5. **IDCODE:** IDCODE register is used to connect from TDI to TDO pin.
6. **CLAMP:** Clamp instruction first uses preload instruction to load values into BSCAN cells and then uses the bypass register to drive data from TDI to TDO.
7. **HIGHZ:** High Z and clamp instructions are almost similar. Here first preload instruction is used to preload control -to- Z values into cells and high Z instruction uses bypass register to shift data from TDI to TDO pin**.**

4.3.5 Resetting The TAP

The TAP and its related hardware can be reset by transitioning the TAP controller to the Test-Logic-Reset state. Transitioning to the Test-Logic-Reset state can happen one of two ways:

- Assert the TRST_N pin at any time. This asynchronously resets the TAP controller.
- Hold the TMS pin high for 5 consecutive cycles of TCK. This is guaranteed to transition the TAP controller to the Test-Logic-Reset state on a rising edge of TCK.

Upon reset, if the slave TAP has an IDCODE associated with it, then this value will be loaded into the IR (parallel register). The TDo pin will be tri-stated. The value of IDCODE can be read to identify the TAP and various other details of the design.

The TDO enable (TDOen) is required on each slave (and secondary slave) TAP interface to allow an integration team the flexibility in using it to drive a CMOS IO pad if necessary. According to the IEEE1149.1 spec, the TDO pad must only be driven when shifting the DR or IR registers, otherwise, it is tri-stated. Generally, the master TAP IO will be constructed such that the TDO drives an open-drain IO pad and therefore, when TDO is not being driven it will automatically revert to a high-impedance state. For the re-usable design principles each agent's TAP must provide the TDO enable just in case such a scenario arises in the future. The PTH slave TAP generates a TDO enable in accordance with the requirements (drive TDO enable high when the TAP FSM is in SHIFTDR or SHIFTIR states).

4.3.6 JTAG Registers

All the registers used for JTAG are shift-register-based to allow data to be written to them (and simultaneously read out) in a serial manner. This serial access is done by first logically connecting one register between the TDI and TDO pins; and then performing the serial shift. The TAP controller handles this procedure. There are two kinds of registers in this standard: the Instruction register and the Test Data registers. Typically, only one Instruction register exists in the TAP block. This register allows the user to load a TAP instruction into the design. The instruction is then used to either select

an internal test to be performed, to specify which test data register to access next or to do both. Test data registers are generally used to control and/or observe internal signals of the design. There are two test data registers that must be present in any JTAG design: the Bypass register and the Boundary-scan register. The Bypass register is a 1-bit shift register used to shorten the shift chain between a TAP's TDI and TDO. This is normally used when multiple TAPs are present in a design and are serially strung together via their respective TDI and TDO pins. The Boundary-scan register is used to drive and observe data on the device pads/pins.

JTAG helps to test the board level connectivity. Also, with the help of its serial shifting protocol, the values of a given register can be read or written. Bypass register is of great use in a system which consists of number of IPs with that many TAPs. This way, it reduces the latency to one clock cycle, which otherwise would have been an entire chain of boundary scan cells.

The Instruction and Test Data registers generally have the structure shown in Figure 3.5. It is a double-flopped register architecture with the first flop stage called the scan flop, and the second flop stage called the parallel flop.

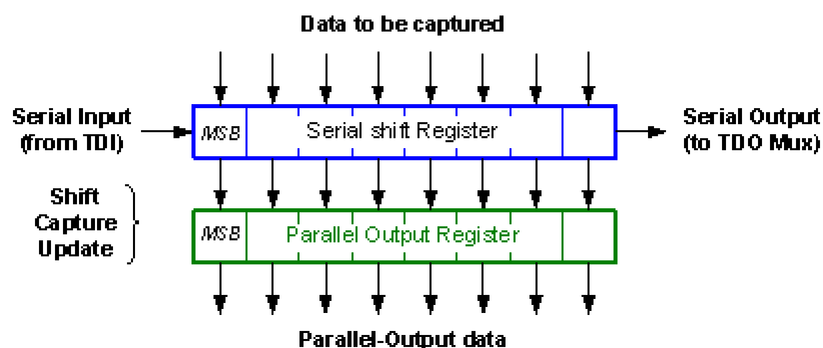


Figure 4.5 JTAG Register Operation

The following 3 operations can be done on these flops: Parallel data capture into the scan flop, Serial data shift into the scan flop through TDI (and out of TDO), and parallel data transfer from the scan flop to the parallel flop. With these operations, the following can be achieved: Parallel-in Parallel-out, Parallel-in Serial-out, Serial-in Parallel-out, Serial-in Serial-out. Of these, the SIPO and PISO operations are most typically used.

4.4 JTAG CRI GLUE LOGIC

CRI protocol dictates how the common register space is to be accessed. It gives out the specification on reading and writing data into the configuration registers. JTAG is the global requester that carries the data that has to be written, and requests for the data that has to be read from a particular register in the register space. Glue logic is responsible for translating the JTAG translations into the format that is required by the CRI master to initiate a transaction on the CRI bus. The JTAG transaction consists of the data as well as the address of the respective register. When the data valid signal is asserted, it indicates that the JTAG data and addresses are ready to be consumed by the CRI master.

CRI slave interacts with the registers in the register space. Whenever there is a request on the CRI slave from the master to read a particular register, the CRI slave accesses those registers and gives the value back to the CRI master which in turn transfers the value to the global requester.

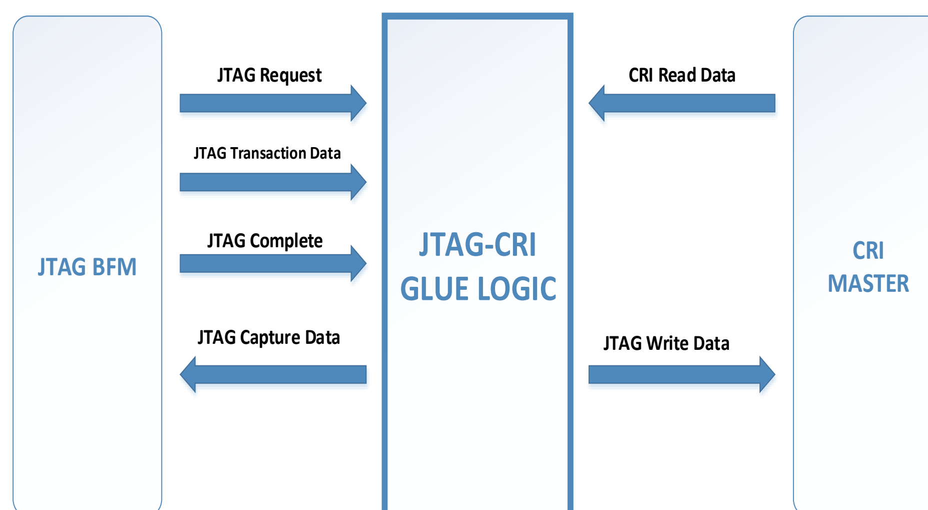


Figure 4.6 JTAG and CRI glue Logic

Similarly, in case of register write, the CRI master hands over the data to be written in the configuration register to the CRI slave which in turn writes in the corresponding register.

JTAG, which is the global requester, communicates with the CRI master. The communication takes place with the assertion of the control signals responsible for the handshaking procedure. JTAG puts a request to the CRI master. A glue logic is required in between which converts the requests from JTAG into a format that is readable by CRI. The CRI responds back by acknowledging the request and further sends the control signals to the slave to perform the required operation.

CHAPTER 5

POST SILICON SUPPORT

5.1. INTRODUCTION

Post Silicon verification is the last step in the validation process of a semiconductor based integrated circuit. In the pre-silicon processes, the devices are validated in a virtual environment. This verification process is done against the design specification using sophisticated simulation and emulation tools. On the contrary, post silicon validation is done on the actual fabricated chips, in the real world environment. Pre silicon environments offer high level of observability, which means that any signal can be observed in the virtual environments. This is however, not the case in a post silicon environment. Additional circuitry needs to be added to aid observability of internal signals. The most important benefit of post silicon validation is that the tests are done on actual fabricated silicon, so the bugs that might have been left in pre silicon validation, get exposed. This makes it an integral part of the verification process. The circuitry that helps in post silicon observation is added to the design and verified in a simulation environment. This is done parallel to the pre silicon functionality verification process.

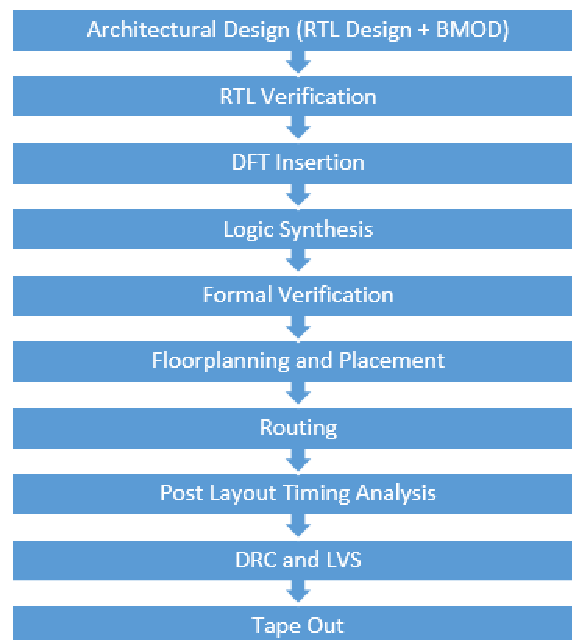


Figure 5.1 Verification Flow

After the first tape out, the silicon prototype is obtained for post silicon verification. It encompasses all the verification effort that is done on the system, before the actual product release. Due to increasing system complexities, there is an emerging need for functional verification at the post

silicon stage, since the coverage provided by pre silicon methodologies is not 100%. The bugs which escape the pre silicon verification are then found in post silicon environment, on the actual test chip. The bugs in this category are often system-level bugs and rare corner-case situations buried deep in the design state space: since these problems encompass many design modules, they are difficult to identify with pre-silicon tools, characterized by limited scalability and performance. Post-silicon validation will screen out of imperfect parts, affirm correct operation on given application, or test robustness of system in the environmental operating conditions. This is particularly important for security gadgets. Customarily this procedure needs a gate level model but working on this new gate level model is a tedious task.

5.2 DESIGN FOR x (DFx)

DFx stands for various verification and testing procedures which are used for complete silicon validation and post silicon validation support.

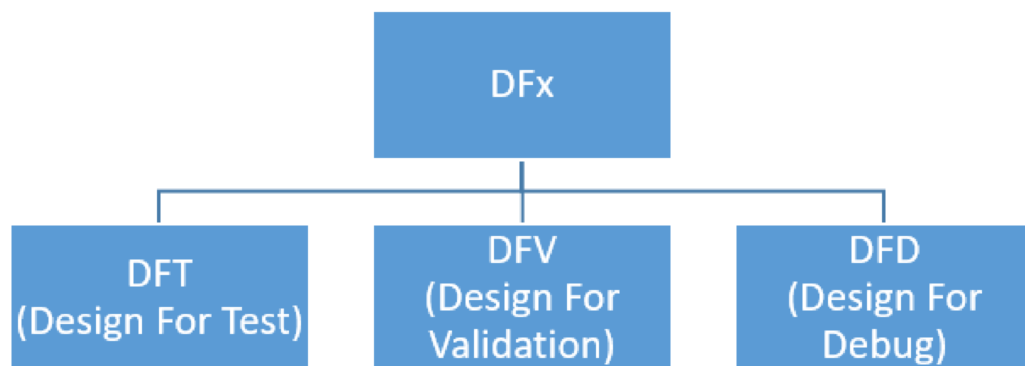


Figure 5.2 Verification Techniques

- **Design For Test:** Design for Test is a technique which facilitates the design to become testable after manufacturing. To run the entire set of functional tests after manufacturing exhaustively is highly cumbersome, and at a point, impractical. This method assures the maturity of a design without the need to run the entire set of tests such that proper manufacturing of the design is ensured. It detects whether the manufactured device is faulty or not. After the post silicon test, if the device is found to be faulty, it is trashed, and not sent to the customer.
- **Design for Validation:** Design for Validation is a technique to ensure the electrical correctness of the design. Several tests such as jitter tolerance, voltage/timing margining, spec

eye compliance are run to ensure that the design is correctly manufactured. Most of the electrical defects targeted during post-silicon validation are first discovered as functional errors. For instance, incorrectly sized transistors on the die may result in unanticipated critical paths. Consequently, when the device is running at high frequency, occasionally data will not propagate through these paths within a single cycle, resulting in incorrect computation results. Similarly, jitter in the clock signal may cause internal flip-flops to latch erroneous data values, or cross-talk between buses may corrupt messages in flight. Such bugs are frequently first found as failures of test sequences to which the prototype is subjected. Designers proceed then to investigate the nature of the bug: by executing the same test sequence on other prototypes they can determine if the bug is of electrical or functional nature that are hard to detect with the limited number of pre silicon simulation cycles and will continue to exist in silicon. The challenges faced in Post-Silicon validation are: Signal Integrity, Design Complexity, and Power and thermal management. These challenges have to be addressed thoroughly in Post-Silicon validation. Due to short post silicon validation cycles, it is important that bugs be detected rapidly. A need for efficient and rigorous post silicon validation methodologies exists and will continue with newer generations of complex designs.

- **Design for Debug:** Design for Debug is a technique with the help of which the faults in the manufactured chips can be debugged. This includes having observability of the internal signals to the output pads of the test chip, such that they can be observed in the post silicon environment. This includes monitor ports in the design onto which the internal signals of interest can be routed to aid the post silicon debug process.

5.3 MONITOR PORT ROUTING

5.3.1 Motivation for JTAG

Before JTAG, the testing of chips was done by the technique of "bed of nails". In this technique, a fixture is used to containing a bed of nails to access the individual devices on the board through test lands laid into the metal interconnects. Then, testing proceeds with first power off and then power on tests. Power off tests check the integrity of the physical contact between the nail and the access point. After this, with the help of calculated impedances, open and short tests are carried out. However, with the increasing complexity, the testing using this technique became difficult due to the gradual loss of physical access. The inability to place a probe on the high speed signals also became a significant problem. Components are now placed on both sides of the chip and with the aforementioned technique, these nails would hit the components. With the reduced spacing between the wires, the nails would short the wires. This led to the need of a built in test delivery system, done by JTAG.

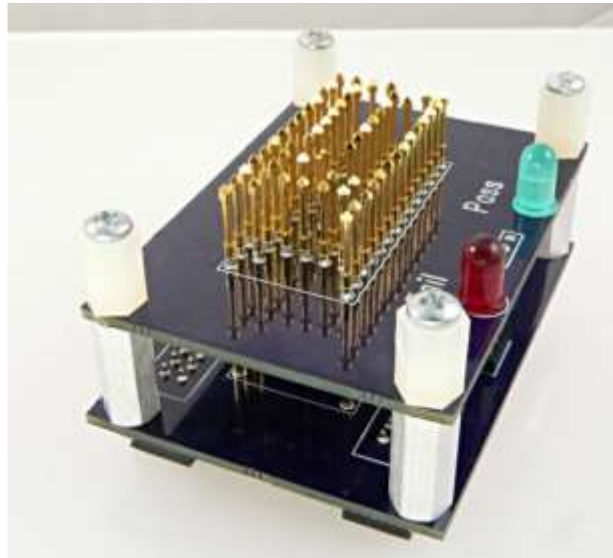


Figure 5.3 Bed of Nails Technique

5.3.2 Signal Routing

If a test fails in silicon, it is very difficult to identify the reason for the failure because of the limited observability. The monitor port routing tests route the digital as well as analog signals of interest to the observation pads. These may include high speed clocks, internal FSM states, signal values, etc., which can be observed in the post silicon environment for debug purposes. With the help of this observation, the underlying cause of the test failures can be identified.

To enable the signals of interest to be observable at the output ports, these signals are fed into the inputs of various multiplexers. These MUXs have select lines, whose values determine which of the input signals will get routed to the output ports. JTAG is used for routing as well as observing these signals. The configuration registers in the register space are accessed and then subsequently read or written with the help of JTAG.

This addition of design architecture for signal routing has to be made such that it does not interfere with the normal functional mode of the design. Signals of interest are taken out and routed via a dedicated path to the observation pads.

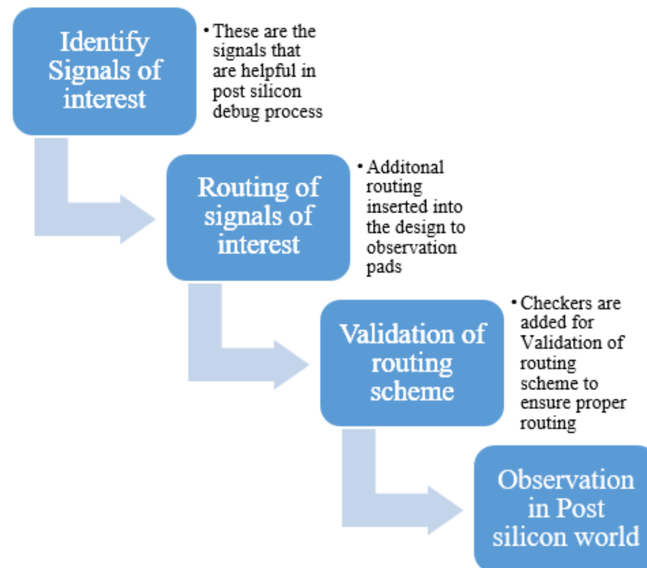


Figure 5.4 Test Flow for Signal Routing

The Design engineers and the DFx architects together decide the signals of interest. This signal list is crucial since it is for aiding post silicon debug. It allows visibility of internal signals which help to root cause the bug.

CHAPTER 6

RESULTS AND OBSERVATIONS

6.1 ASSERTION BASED VERIFICATION

Assertion based verification was done for JTAG protocol. Assertions were written to verify the JTAG FSM discussed in Chapter 3. Further, assertions were written to ensure that the internal configuration registers were being properly read and written. The FSM states change with respect to the input control signal TMS sampled at the JTAG clock, TCK.

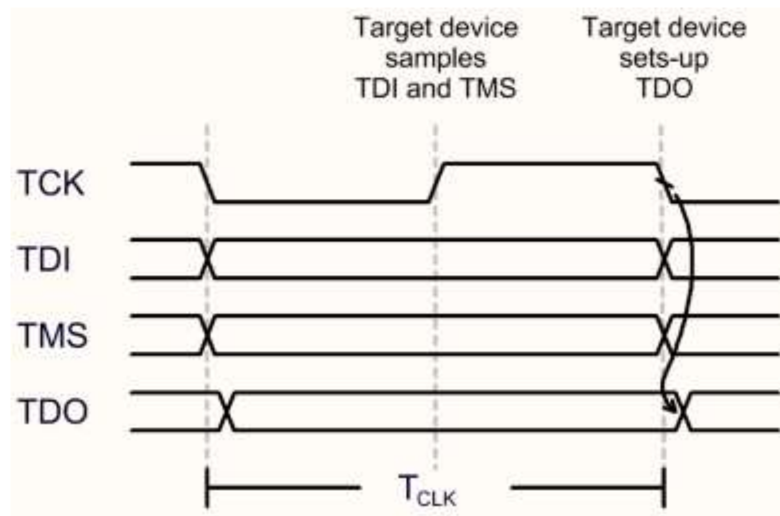


Figure 6.1 Control Signals in JTAG

Assertions were written to ensure that the data sampled as TDO complied with the TMS and TDI signals. Further, assertions were written to ensure proper reading of the internal configuration registers was happening or not.

Whenever there is request to read the value of a particular register, JTAG asserts it on the CRI master. The CRI master then reads the values from the CRI slave, which has access to the internal configuration registers which are desired to be read. The CRI clave returns the read value to the CRI master which further is connected to the JTAG requesting the read.

Assertion based verification is very useful, since it involves static checking of the design. The formal verification tool used drives the values to all the possible conditions such that corner cases are not left out. Formal property verification method explores all the states and transitions in the model with the help of domain specific abstraction techniques such that the entire set of states are executed and exercised to be validated in a single operation.

An assertion is basically a "statement of fact" or "claim of truth" written in System Verilog made about a design by a designer or verification engineer. An engineer will assert or "claim" that certain conditions are always true or never true about a design. If that claim can ever be proven false, then the assertion fails. A property is basically a rule that will be asserted (enabled) to passively test a design. The property can be a simple Boolean test regarding conditions that should always hold true about the design, or it can be a sampled sequence of signals that should follow a legal and prescribed protocol, here, JTAG.

Following is a failing case scenario where the expected value to be read does not match with the actual value of the register. Assertions bring out the corner case scenarios in which the RTL dependencies and transitions are clearly visible to the engineer. These case scenarios might be present deep into the design such that the dependencies might not be apparent to the verification engineer. Assertions clearly state the intent of verification, what is expected out of the design, and when it is expected. This adds additional benefits since a system design having multiple modules operating on different clock frequencies can be easily verified stating the respective clocks with respect to which the assertions are evaluated.

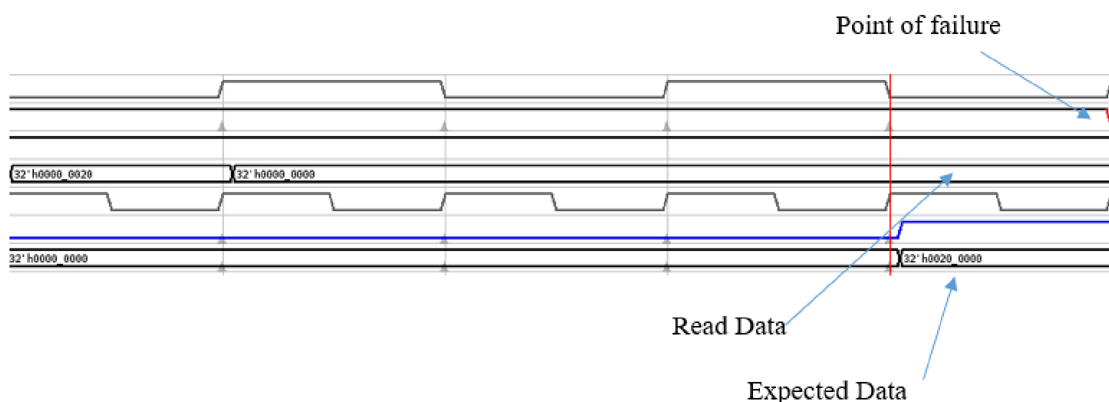


Figure 6.2 Waveforms of a Failing Assertion

Similarly, assertions were written for all the state transitions of the JTAG FSM. The assertions were written to verify properties written in accordance with the design specifications. These properties define the crucial aspects of the design that the RTL should always follow. In case of the assertions written for the glue logic and register space access, these properties implement the proper handshake between the CRI master and slave units with respect to the requests to read or write the internal configuration registers placed by JTAG.

Figure 6.3 Assertions Results in Formal Tool Checker

Assert

Cover (related)

Assert

Cover (related)

Figure 6.4 Significance of Cover Property

The significance of assertions also lies in its cover property. Whenever the antecedent of the assertion property is satisfied, only then the cover will hit. This gives high reliability, since the absence of any errors does not signify that the design in bug free. High coverage ensures that all the properties evaluating a particular design cover all its features. For an assertion to pass, first its cover property must also pass, indicating that the design has the required pre conditions that we are willing to check. Then, when the cover is hit, the tool analyzes all the dependencies throughout the design and evaluates them by assigning all possible values to the controls. In such a way, it is ensured that all the possible scenarios, have been “covered”. This in turn provides the information on how deep the tool has gone inside the design to evaluate all the possible scenarios.

The dependence of a signal can be brought out with the help of formal checks. The tool tries to find out what all these dependencies are and then drives them into various values to make the assertion fail. The tool follows a pessimistic approach wherein it tries to make the assertion fail by randomly and exhaustively driving all the inputs through all the set of values that they can take.

6.2 TESTS FOR POST SILICON SUPPORT

Test cases were written to aid post silicon debug. These tests were monitor tests written in the Open Verification Methodology environment. The test sequence included the necessary routing code to route the signals, high speed clocks to the observation ports. These included the routing of the desired signals via different MUXing schemes to the destination pads. Checkers were written to ensure that the signals on the observation pads match the source signals being routed.

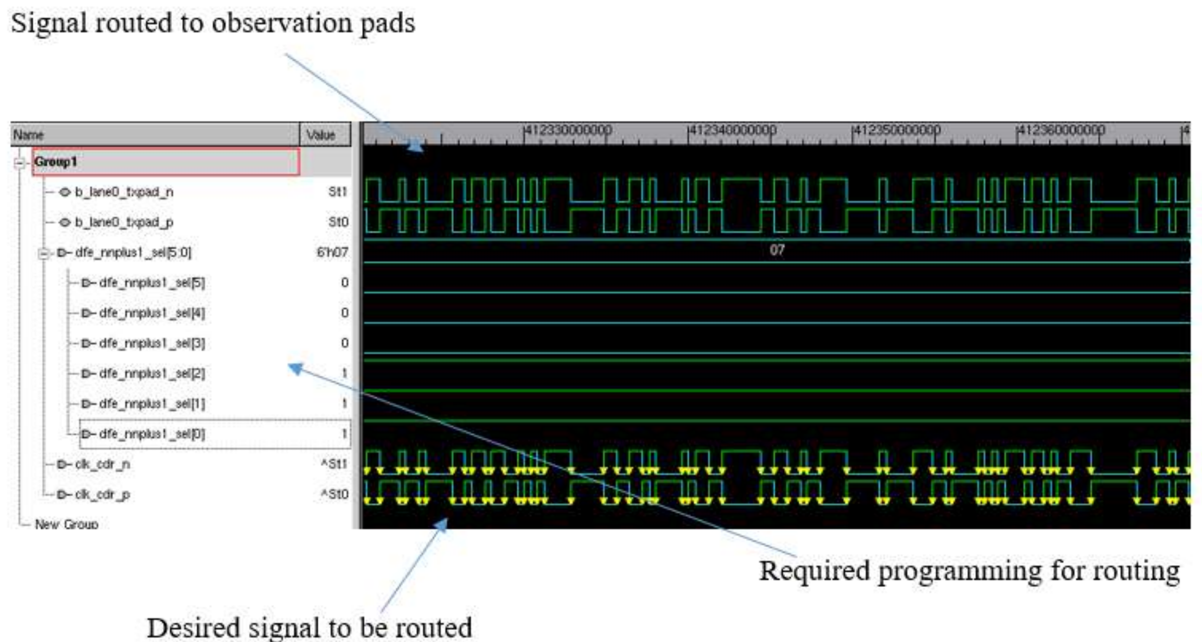


Figure 6.5 Simulation Waveforms of Routing Tests

There are a number of signals which need to be observed in a post silicon debug process. Hence, a lot of work goes into deciding the signals which will have post silicon observability. Design engineers as well as DFx architects prepare these signal lists together keeping into consideration all these requirements. Since the designs are very complex in nature, there can be a number of issues due to which a test chip might not run during power on. These issues can be debugged with the help of these internal signal visibility. For any failing operation, if the corresponding signals that play the crucial role in bringing up that functionality have observability, the error can be root caused. Therefore, the signals selection is an important part in post silicon debug. In the silicon, the observation pads can be probed and the signals can be viewed on the oscilloscope.

6.3 REGRESSION ANALYSIS

Despite significant advances in the scope and sophistication of EDA tools, the percentage of design re-spins caused by functional errors is rising. This situation is due in part to escalating system complexity and in part to the increased use of third-party intellectual property [21]. When an entire

system is integrated, validation needs to be done for all the design iterations on the complete system. Once a block has been verified, after integration, it needs to be verified again, extensively, on the entire system. This way, the bugs that might be caused due to integration alone, and not in any individual block can be identified.

Regression testing is a well-known verification technique in which continuous re-testing of the design is done throughout the design modification period, to ensure that an added functionality, or even a little design change does not cause any previously passing verification test to fail. In this process, the same set of tests are run again and again on the design to ensure that the added features have not introduced any new errors. Regressions analysis consists of tests written against the design behavior which capture all the aspects of the design intention.

Regression testing can be further of two types, dynamic and static. Dynamic Regression analysis is done with the help of digital simulation techniques and static analysis is done with the help formal checking tools.

In the project, many of such tests to ensure proper routing of desired signals were written. Regression analysis was done for these tests to ensure that any design changes does not affect the routing scheme of these signals. The number of tests was very big, implying the added observability of many signals, FSM states, clocks, etc.

Figure 6.6 Regression Analysis

The process of Regressions analysis starts as soon as the verification of the design starts. Few basic tests are first coded and a regression is run on them to ensure basic features. With the course of time,

as the verification moves further, other tests are added into the regression lists. These regression lists contain all the tests that need to be run periodically on the design.

Regression automatization was also done with the help of “Perl” scripts. Running regressions manually daily can be a tiring task. It also poses the threat for a possible human error. Regressions are a way to ensure that the added functionality and other design changes do not affect the previously passing test cases. To ensure this, regressions need to be run continuously to catch any failing test case scenario with the rapidly changing RTL. The entire process of regressions was also automatized.

CHAPTER 7

CONCLUSION AND FUTURE SCOPE

The project highlights the verification methodologies used in the industry. Formal tools based on mathematical models using System Verilog Assertions have been written to ensure proper reading and writing of internal configuration registers in the design. Register space in the design can be accessed by following an industry specific protocol and assertions have been written to verify the protocol.

Further, post silicon support has been added in the design to aid post silicon debug process. This additional circuitry provides observability to the internal signals after tape out. This in turn helps in debugging the errors that are observed in the manufactured silicon. This has been done with the help of OVM methodology, writing checkers to ensure proper routing of these internal signals to the observation pads.

Synthesis of assertions is needed for post-synthesis checks. Assertion based checks in this report have been done in pre silicon simulation environment on RTL abstraction level. There is always a chance to find a functional bug after synthesis, placement or routing. So along with RTL, assertions can be synthesized that can check the functional correctness and coverage at post placement and routing phase.

Assertions can also be written to specify the power intent of a design. Assertions should be automatically generated from reading design and UPF. Since all the assertions are based on properties defined for the design and its power intent, it can be automatized to reduce human error and save time.

REFERENCES

[1] Bening, L. and Foster, H., 2001. *Principles of verifiable RTL design* (pp. 239-245). Springer US.

[2] Glasser, Mark. *Open verification methodology cookbook*. Springer Science & Business Media, 2009.

[3] Rashinkar, P., Paterson, P. and Singh, L., 2007. *System-on-a-chip verification: methodology and techniques*. Springer Science & Business Media.

[4] Ghenassia, Frank, and Alain Clouard. "Tlm: An overview and brief history." In *Transaction Level Modeling with SystemC*, pp. 1-22. Springer, Boston, MA, 2005.

[5] Brahme, Dhanajay, et al. "The transaction-based verification methodology." *Cadence Design Systems, Inc* (2000).

[6] Kasuya, Atsushi, and Tesh Tesfaye. "Verification methodologies in a TLM-to-RTL design flow." In *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pp. 199-204. IEEE, 2007.

[7] Phebe, Bonum Sherley, and J. Anusha. "DEVELOPMENT, TESTING AND VERIFICATION OF IP CORE FOR SYNCHRONOUS SERIAL PORT (S-PORT) INTERFACE USING OVM." (2016).

[8] Spear, Chris. *SystemVerilog for verification: a guide to learning the testbench language features*. Springer Science & Business Media, 2008.

[9] Mehta, Ashok B. "System verilog assertions." In *SystemVerilog Assertions and Functional Coverage*, pp. 9-28. Springer, New York, NY, 2014.

[10] August, Nathaniel. "A robust and efficient pre-silicon validation environment for mixed-signal circuits on intel's test chips." In *Quality Electronic Design, 2008. ISQED 2008. 9th International Symposium on*, pp. 423-428. IEEE, 2008.

[11] Kannavara, R., 2013, July. Towards a unified framework for pre-silicon validation. In *Information, Intelligence, Systems and Applications (IISA), 2013 Fourth International Conference on* (pp. 1-7). IEEE.

[12] Adir, A., Copty, S., Landa, S., Nahir, A., Shurek, G., Ziv, A., Meissner, C. and Schumann, J., 2011, March. A unified methodology for pre-silicon verification and post-silicon validation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011* (pp. 1-6). IEEE.

[13] Nicolici, Nicola, and Ho Fai Ko. "Design-for-debug for post-silicon validation: Can high-level descriptions help?." In *High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*, pp. 172-175. IEEE, 2009.

[14] Ma, S., 2015. High quality functional coverage based trace signal selection for post-silicon validation.

[15] Lei, T., He, H. and Sun, Y., 2009, October. ISTA: An embedded architecture for post-silicon validation in processors. In *ASIC, 2009. ASICon'09. IEEE 8th International Conference on* (pp. 593-596). IEEE.

[16] Detraz, S., Sigaud, C., Seif El Nasr, S., Papakonstantinou, I., Papadopoulos, S., Versmissen, H., Moreira, P., Soos, C., Stejskal, P., Silva, S. and Troska, J., 2009. FPGA-based bit-error-rate tester for SEU-hardened optical links.

[17] Sadasivam, S.K., Alapati, S. and Mallikarjunan, V., 2012, September. Test generation approach for post-silicon validation of high end microprocessor. In *Digital System Design (DSD), 2012 15th Euromicro Conference on* (pp. 830-836). IEEE.

[18] Wang, D., Jiang, P.H., Kukula, J., Zhu, Y., Ma, T. and Damiano, R., 2001, June. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proceedings of the 38th annual Design Automation Conference* (pp. 35-40). ACM.

[19] Kuehlmann, A., Paruthi, V., Krohm, F. and Ganai, M.K., 2002. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *21*(12), pp.1377-1394.

[20] Dasgupta, P., 2006. A roadmap for formal property verification. In *A Roadmap for Formal Property Verification* (pp. 217-241). Springer, Dordrecht.

[21] Burgess, Ian, "*Regressions Testing: Gate Level Functional Verification is Imperative and Equivalence Checking Provides a Solution*", Mentor Graphics Corp.

[22] Wong, W.E., Horgan, J.R., London, S. and Agrawal, H., 1997, November. A study of effective regression testing in practice. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on* (pp. 264-274). IEEE.

[23] Fadiheh, M.R., Urdahl, J., Nuthakki, S.S., Mitra, S., Barrett, C., Stoffel, D. and Kunz, W., 2018, March. Symbolic quick error detection using symbolic initial state for pre-silicon verification. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018* (pp. 55-60). IEEE.

[24] Foster, H.D., 2015, June. Trends in functional verification: a 2014 industry study. In *Proceedings of the 52nd Annual Design Automation Conference* (p. 48). ACM.

[25] Mehta, A.B., 2018. Functional Verification: Challenges and Solutions. In *ASIC/SoC Functional Design Verification* (pp. 5-12). Springer, Cham.

[26] Herdt, V., Le, H.M., Große, D. and Drechsler, R., 2018, March. Towards fully automated TLM-to-RTL property refinement. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018* (pp. 1508-1511). IEEE.